Enrolment No./Seat No_______________

GUJARAT TECHNOLOGICAL UNIVERSITY

BE - SEMESTER–IV EXAMINATION – SUMMER 2025

Subject Code: 3140707 **Date:15-05-2025**

Subject Name: Computer Organization & Architecture

Time: 10:30 AM TO 01:00 PM **Total Marks:70**

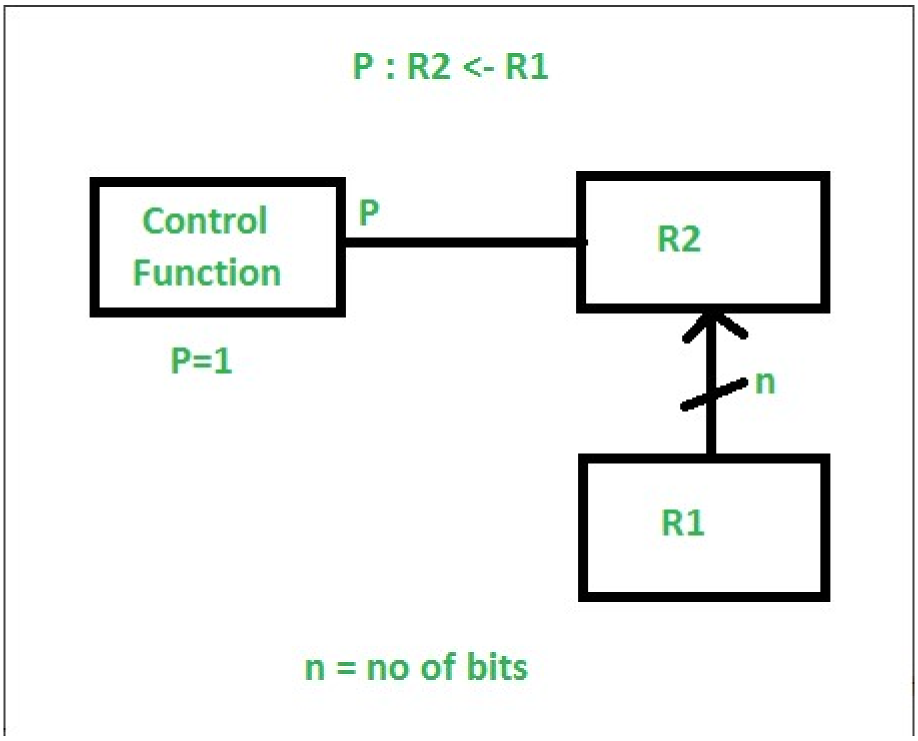
Instructions:

1. **Attempt all questions.**
2. **Make suitable assumptions wherever necessary.**
3. **Figures to the right indicate full marks.**
4. **Simple and non-programmable scientific calculators are allowed.**

			MARKS
Q.1	**(a)**	Explain the Register Transfer Language with block diagram.	**03**
	(b)	Explain three state bus buffer.	**04**
	(c)	Explain shift micro operations and draw 4-bit combinational circuit shifter.	**07**
Q.2	**(a)**	List and explain any three register reference instruction.	**03**
	(b)	Explain instruction format with its types.	**04**
	(c)	Draw and explain Common Bus System for basic computer register.	**07**
		OR	
	(c)	Explain the basic working principle of the Control Unit with timing diagram.	**07**
Q.3	**(a)**	List out any three register of basic computer.	**03**
	(b)	State various phases of instruction cycle.	**04**
	(c)	Write an assembly level program to find average of 10 numbers stored at consecutive location in memory.	**07**
		OR	
Q.3	**(a)**	Convert following hexadecimal number into decimal, octal and binary. 1) 4A	**03**
	(b)	Explain any 4 addressing modes with example.	**04**
	(c)	What is an Interrupt Cycle? Draw and explain flow chart of it.	**07**
Q.4	**(a)**	Explain register stack.	**03**
	(b)	Write an assembly language program to Add two double precision numbers.	**04**
	(c)	Explain the working of Second Pass Assembler with its flowchart.	**07**
		OR	
Q.4	**(a)**	What is address sequencing?	**03**
	(b)	Write short note on subroutine.	**04**
	(c)	Draw and explain flow chart for multiplication program.	**07**
Q.5	**(a)**	Explain various types of interrupts.	**03**
	(b)	What are status register bits? Draw and explain the block diagram showing all status registers.	**04**
	(c)	Write a note on asynchronous data transfer.	**07**
		OR	
Q.5	**(a)**	What is Memory Interleaving?	**03**
	(b)	Differentiate RISC and CISC.	**04**
	(c)	Explain Flynn's classification for computers.	**07**

****

Q.1 (a) Explain the Register Transfer Language with block diagram.



Register Transfer Language (RTL)

Register Transfer Language (RTL) is a symbolic notation used to describe **micro-operations** and the **data flow between registers** inside a digital system or CPU.
It explains **what data is transferred**, **between which registers**, and **under what control condition** in a single clock cycle.

In simple words, RTL tells:

- **Which register sends data**
- **Which register receives data**
- **When the transfer happens (control signal)**

Basic RTL Statement

A general RTL statement is written as:

```
P : R2 ← R1
```

Meaning:

- When **control signal P = 1**
- The contents of **register R1** are transferred to **register R2**
- Transfer occurs in **one clock pulse**

- Width of transfer depends on **number of bits (n)**

Explanation of the Diagram

1. Control Function

- Generates control signal **P**
- When **P = 1**, data transfer is enabled
- Correctly represents conditional transfer

2. Source Register (R1)

- Holds **n-bit data**
- Acts as the **data source**
- Data flows upward from R1

3. Destination Register (R2)

- Receives data from R1
- Enabled only when control signal **P** is active

4. Data Path

- The line between **R1 and R2** represents an **n-bit data bus**
- Label **n = number of bits** is correctly mentioned

RTL Operation (Step-by-Step)

1. Control unit evaluates the condition **P**
2. If **P = 1**, control signal activates
3. Data stored in **R1** is placed on the bus
4. **R2** loads the data at the next clock edge
5. If **P = 0**, no transfer occurs

Micro-operation Representation

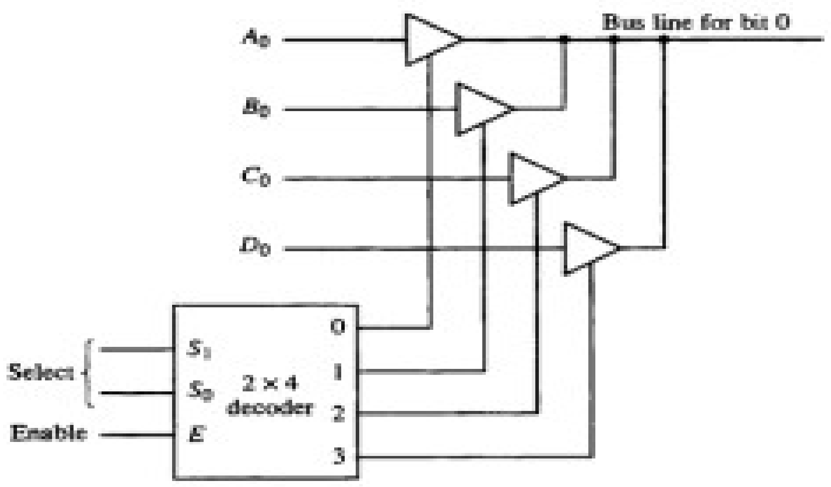
This diagram represents a **register transfer micro-operation**, not an arithmetic or logic operation.

Example micro-operations:

- `R2 ← R1` → Data transfer

- `R3 ← R1 + R2` → Arithmetic
- `R1 ← R1 AND R2` → Logical

(b) Explain three state bus buffer.



Q.1 (b) Explain Three-State Bus Buffer

What is a Three-State Bus Buffer?

A **three-state bus buffer** is a digital circuit used in computer systems to connect multiple registers or devices to a **common bus**.
Unlike normal logic devices that have only two output states (**0** and **1**), a three-state buffer has **three output states**:

1. Logic **0**
2. Logic **1**
3. **High-Impedance (Hi-Z)** state

The **High-Impedance** state electrically disconnects the device from the bus.

Why Three-State Buffers Are Needed

In a system where many registers share a **single data bus**, only **one register** is allowed to place data on the bus at a time.
If more than one register drives the bus simultaneously, it causes **bus contention** and data corruption.

Three-state buffers solve this problem by:

- Allowing only the selected register to drive the bus

- Forcing all other registers into **Hi-Z** state

Basic Operation

A three-state buffer has:

- **Data input**
- **Enable (control) signal**
- **Output connected to bus**

Enable	Input	Output
0	X	High-Impedance (Hi-Z)
1	0	0
1	1	1

- When **Enable = 1**, buffer behaves like a normal buffer
- When **Enable = 0**, output goes to **Hi-Z** state

Working in a Bus System

1. Control unit activates **only one enable signal**
2. Selected register's buffer drives the bus
3. All other buffers remain in **Hi-Z** state
4. Destination register reads data from the bus
5. On next clock pulse, data transfer completes

Advantages of Three-State Bus Buffer

- Prevents **bus contention**
- Reduces **hardware complexity**
- Allows multiple devices to share one bus
- Simplifies CPU and register interconnections
- Improves system reliability

Applications

- CPU internal data buses
- Register-to-register transfer
- Memory and I/O interfacing
- Microprocessor systems

(c) Explain shift micro operations and draw 4-bit combinational circuit shifter.

Shift Micro-operations

Shift micro-operations move the binary contents of a register **left or right** by one or more bit positions.
They are commonly used in **arithmetic operations (×2, ÷2)**, **data alignment**, and **serial data processing**.

In RTL, shift micro-operations are written as:

- **Logical shift**
- **Arithmetic shift**
- **Circular (rotate) shift**

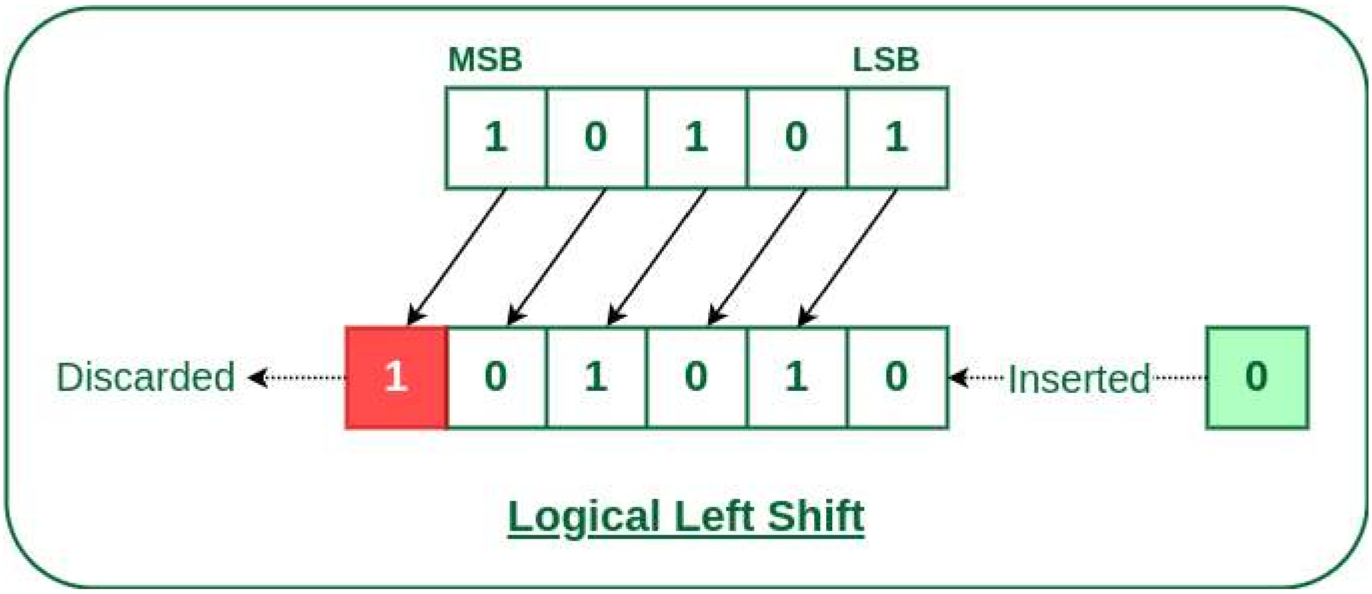
Logical Shift

A **logical shift micro-operation** moves the bits of data in a register either **left or right**, while **filling the vacant bit positions with zeros**. It is mainly used in **binary arithmetic** and **data processing operations**.

There are **two types of logical shift micro-operations**:

1) Logical Left Shift

In a logical left shift, all bits are shifted **one position to the left**.
The **least significant bit (LSB)** is filled with **0 (serial input)**, while the **most significant bit (MSB)** is **discarded**.
The left shift operator is denoted by <<.



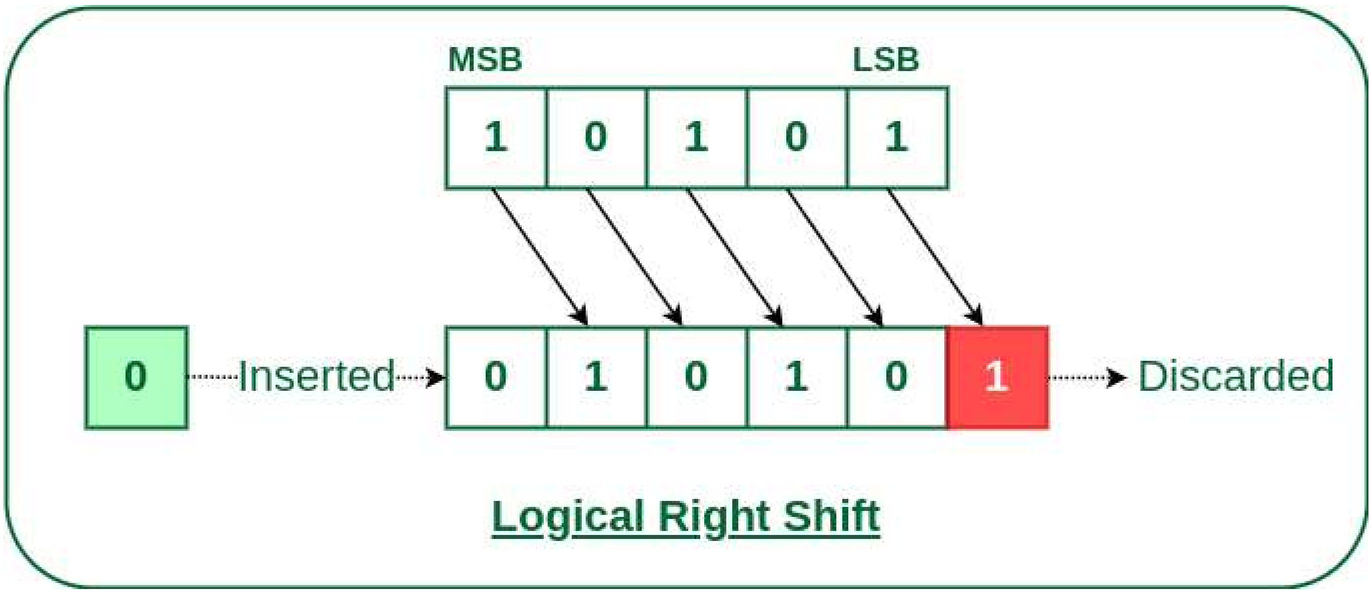
Usage: Used in multiplication of unsigned binary numbers.

Example: *Let's take an 8-bit unsigned binary number 01010011 (which is 83 in decimal). If we perform a logical shift left:*

- ***Before Shift:*** *01010011 (83 in decimal)*
- ***After Logical Shift Left:*** *10100110 (166 in decimal)*

2) Logical Right Shift

In a **logical right shift**, all bits are shifted **one position to the right**.
The **least significant bit (LSB)** is **discarded**, and the **most significant bit (MSB)** is filled with **0**.
The right shift operator is denoted by >>.



Usage: Used in division of unsigned binary numbers.

Example: *Let's take the same 8-bit binary number 01010011 (83 in decimal). If we perform a logical shift right:*

- ***Before Shift:*** *01010011 (83 in decimal)*
- ***After Logical Shift Right:*** *00101001 (41 in decimal)*

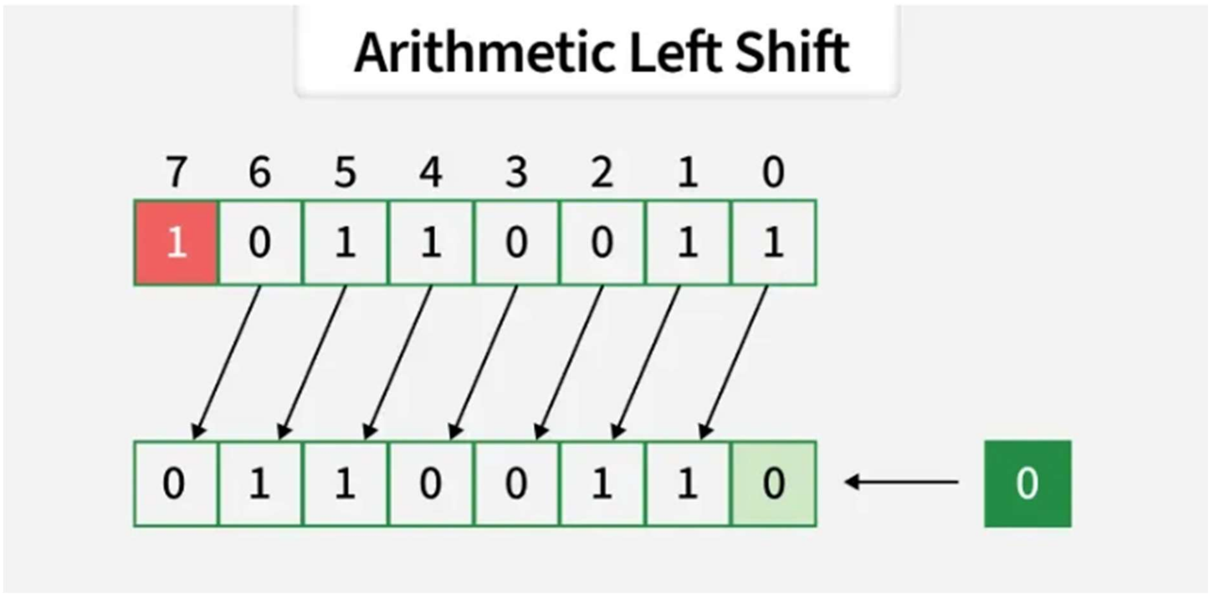
Arithmetic Shift

Arithmetic Shift

An **arithmetic shift micro-operation** shifts a **signed binary number** either **left or right**.
There are **two types** of arithmetic shifts.

1) Arithmetic Left Shift

In an arithmetic left shift, all bits are shifted **one position to the left**.
The **least significant bit (LSB)** is filled with **0**, and the **most significant bit (MSB)** is **discarded**, similar to a logical left shift.
However, it differs conceptually because an arithmetic left shift is interpreted as **multiplication by 2** for **signed integers**, while a logical left shift does not inherently imply a numeric meaning.



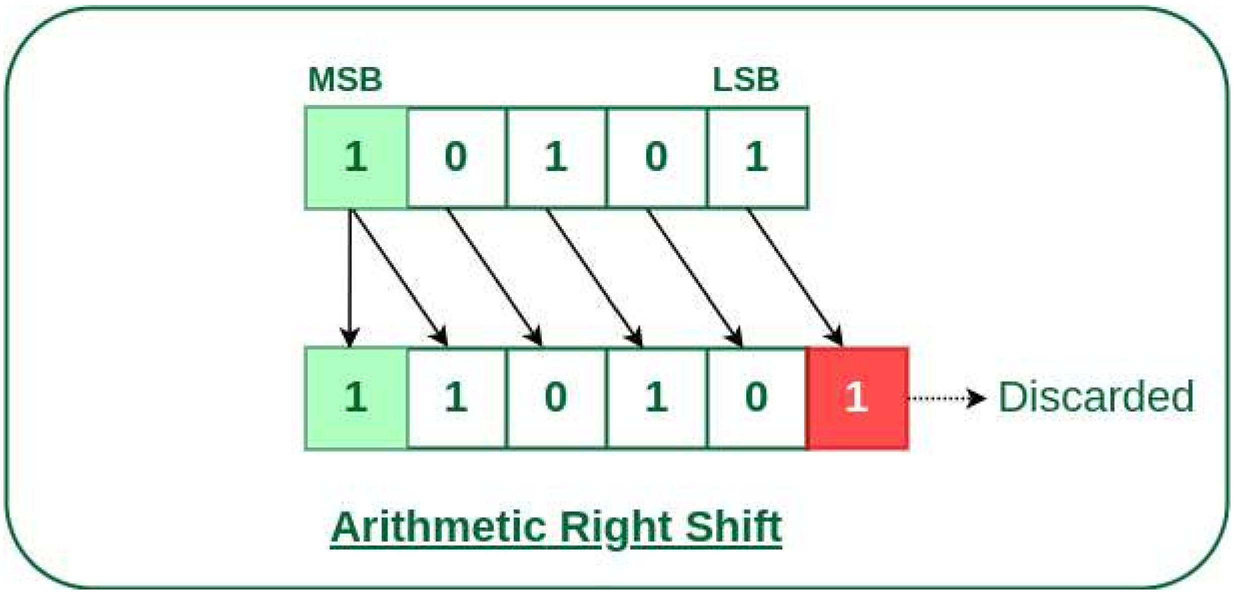
Usage: Used for multiplying signed binary numbers by powers of 2.

Example: *Let's take a signed 8-bit binary number in two's complement form: 11111111 (which represents -1 in decimal). If we perform an arithmetic shift left:*

- ***Before Shift:*** *11111111 (-1 in decimal)*
- ***After Arithmetic Shift Left:*** *11111110 (-2 in decimal)*

2. Arithmetic Right Shift

In this shift, each bit is moved to the right one by one and the least significant(LSB) bit is rejected and the empty most significant bit(MSB) is filled with the value of the previous MSB.



Arithmetic Right Shift

Usage: Used for dividing signed binary numbers by powers of 2.

Example: *Let's take a signed 8-bit binary number in two's complement form: 11111111 (which represents -1 in decimal). If we perform an arithmetic shift right:*

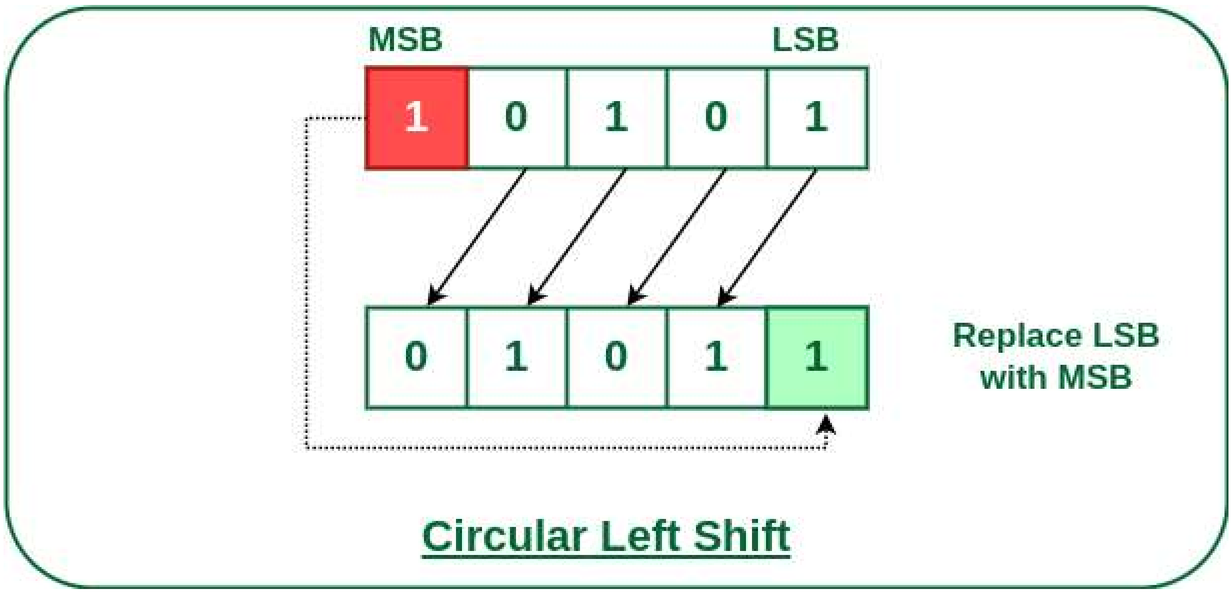
- ***Before Shift:*** *11111111 (-1 in decimal)*
- ***After Arithmetic Shift Right:*** *11111111 (-1 in decimal)*

Circular Shift

The circular shift circulates the bits in the sequence of the register around both ends without any loss of information. Following are the two ways to perform the circular shift:

1. Circular Left Shift

In this micro shift operation each bit in the register is shifted to the left one by one. After shifting, the LSB becomes empty, so the value of the MSB is filled in there.



Circular Left Shift

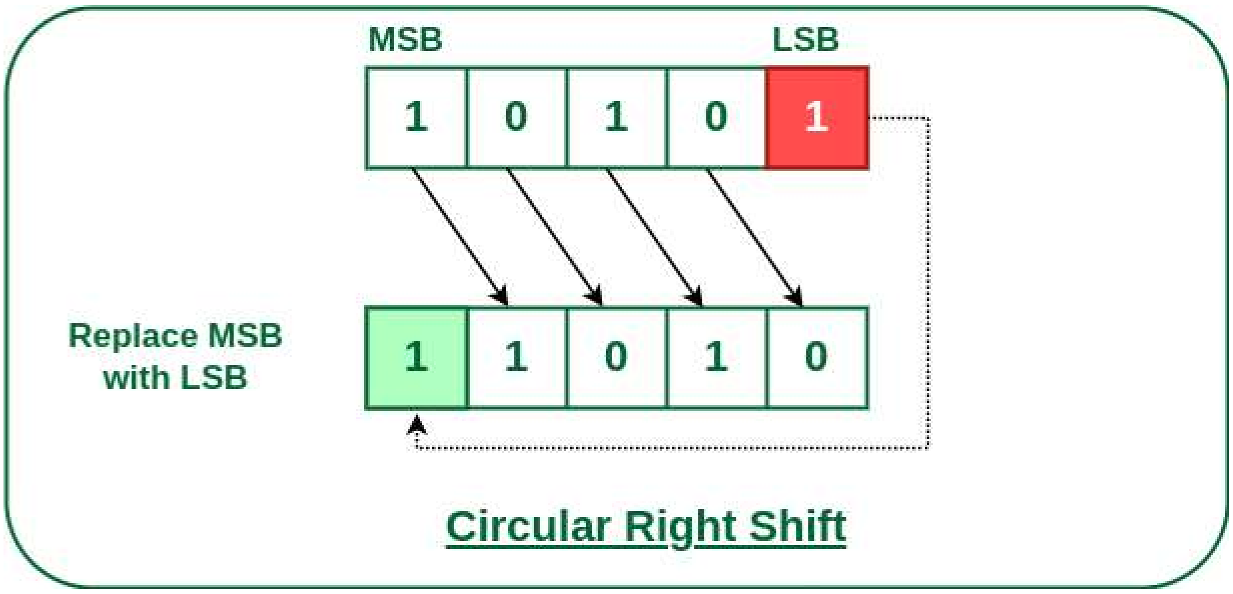
Usage: Used in data encryption algorithms and certain arithmetic operations, where the shift should be cyclic.

Example: *Let's take an 8-bit binary number 11010011. If we perform a circular shift left:*

- ***Before Shift:*** *11010011*
- ***After Circular Shift Left:*** *10100111*

2. Circular Right Shift

In this micro shift operation each bit in the register is shifted to the right one by one. After shifting, the MSB becomes empty, so the value of the LSB is filled in there.

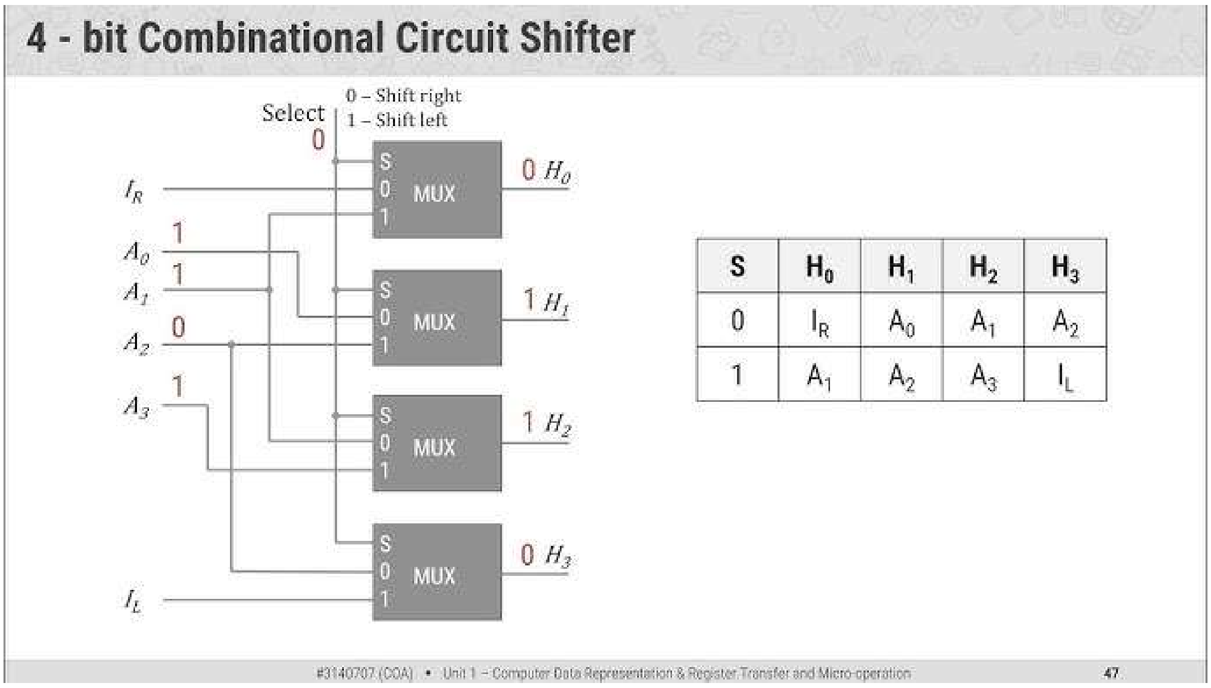


Usage: Used in situations where rotation of bits is required, such as in encryption algorithms.

Example: *Let's take the same 8-bit binary number 11010011. If we perform a circular shift right:*

- ***Before Shift:*** *11010011*
- ***After Circular Shift Right:*** *11101001*

4-bit combinational circuit shifter



4 - bit Combinational Circuit Shifter

S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

#3140707 (COA) • Unit 1 – Computer Data Representation & Register Transfer and Micro-operation 47

Q.2 (a) List and explain any three register reference instruction.

Register reference instructions operate directly on the **contents of registers** (mainly the accumulator) and do **not require a memory address**. These instructions are executed in **one clock cycle** and are commonly used in basic computer architecture.

Below are **any three register reference instructions**:

---

1) CLA – Clear Accumulator

- This instruction clears the contents of the accumulator.
- All bits of the accumulator are set to **0**.
- Used to initialize the accumulator before operations.

Operation:

```
AC ← 0
```

---

2) CMA – Complement Accumulator

- This instruction complements (inverts) each bit of the accumulator.
- Converts **0 to 1** and **1 to 0**.
- Useful in arithmetic and logic operations.

Operation:

```
AC ← AC'
```

---

3) INC – Increment Accumulator

- This instruction increases the value stored in the accumulator by **1**.
- Used in counting and looping operations.

Operation:

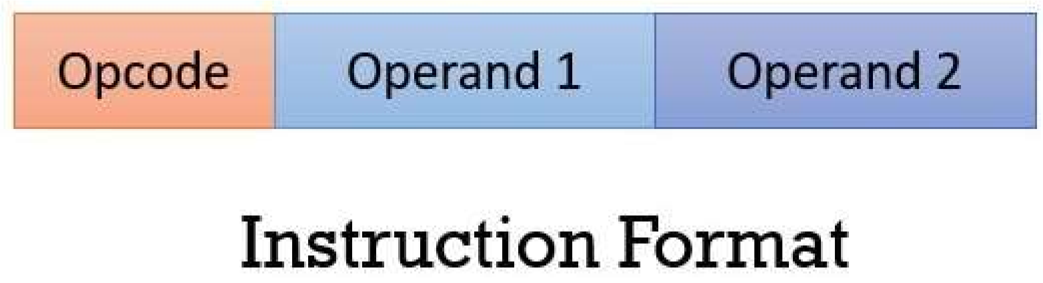
```
AC ← AC + 1
```

(b) Explain instruction format with its types.

Instruction Format

An **instruction format** defines the **layout of bits** in a machine instruction.
It specifies **what operation is to be performed** and **where the operands are located**.
Each instruction generally consists of an **opcode field** and one or more **operand/address fields**.

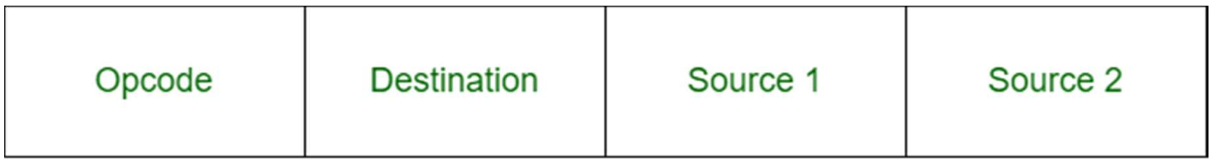
General Instruction Format



- **Opcode**: Specifies the operation (ADD, SUB, LOAD, etc.)
- **Operand/Address field**: Specifies memory location, register, or immediate data

1) Three-Address Instruction Format

In a **three-address instruction format**, the instruction contains **three address fields**: two for source operands and one for the destination.
All operands are explicitly specified, which reduces the number of instructions required for computation.



3-address Instruction Format

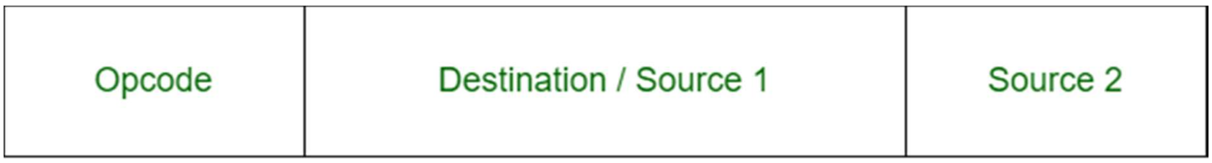
Example:

ADD R1, R2, R3 → R1 ← R2 + R3

Here,
Data from Source 1 and Source 2 is used for the operation, and the result is stored in the Destination register.

2) Two-Address Instruction Format

In a **two-address instruction format**, the instruction has **two address fields**.
One of the operands also acts as the destination for the result.

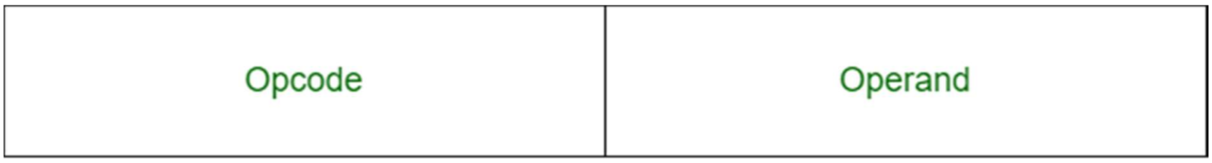


2-address Instruction Format

ADD R1, R2 → R1 ← R1 + R2

3) One-Address Instruction Format

In a **one-address instruction format**, the instruction contains **only one address field**.
The other operand is assumed to be in an **implicit accumulator (AC)**.



1-address Instruction Format

ADD X → AC ← AC + M[X]

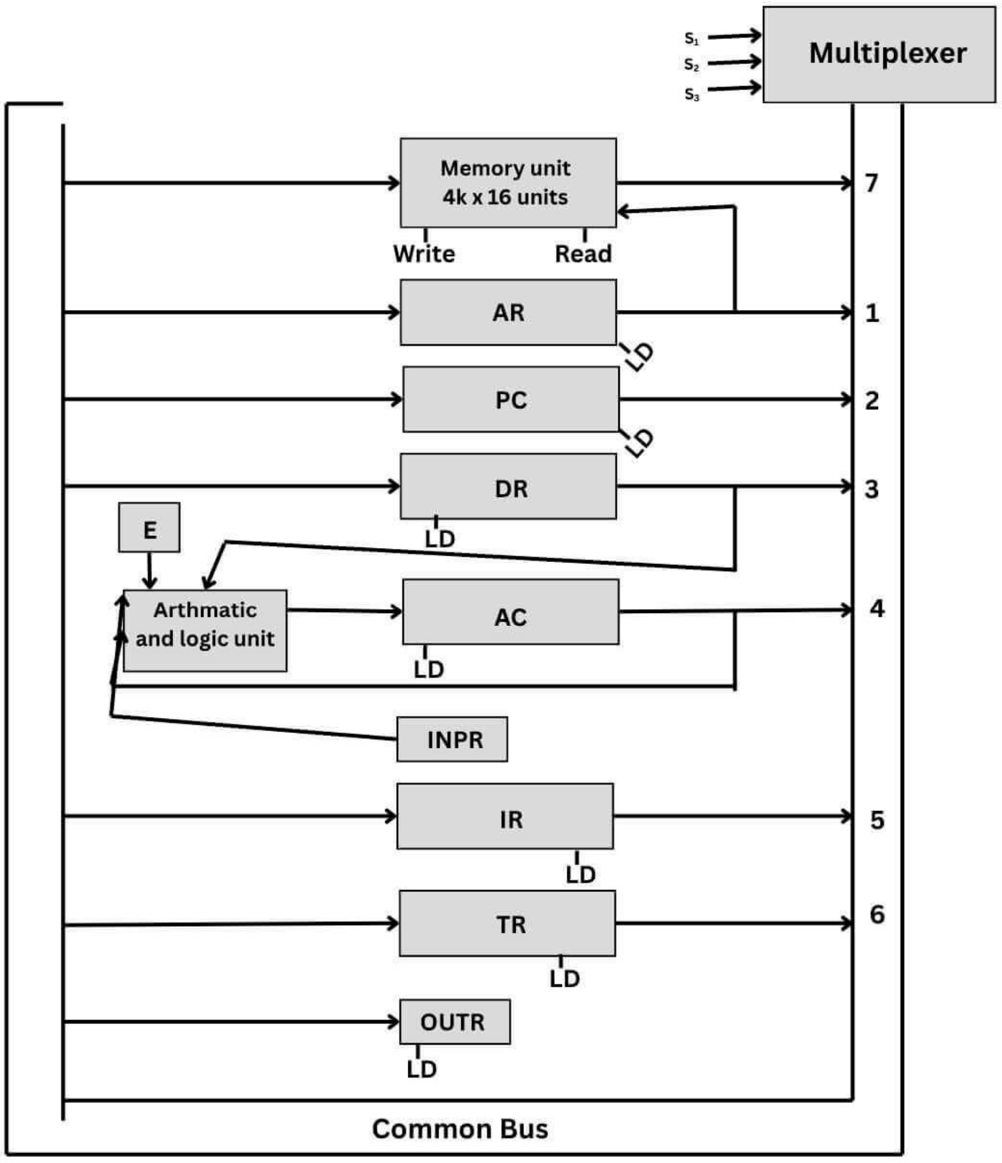
4) Zero-Address Instruction Format

In a **zero-address instruction format**, the instruction contains **no address fields**.
Operands are implicitly taken from the **stack**.

EXAMPLE : **ADD**

0-address Instruction Format

(c) Draw and explain Common Bus System for basic computer register.



Common Bus System

Common Bus System for Basic Computer Registers

A **common bus system** is an interconnection scheme used in a basic computer to transfer data among **registers, memory, and the arithmetic logic unit (ALU)** using a **single shared data bus**.
Instead of having separate point-to-point connections between every register, all registers communicate through this common bus, which significantly **reduces hardware complexity, wiring, and cost**.

Need for Common Bus System

In a computer with many registers, direct connections between each register would require a large number of wires and control lines.
The common bus system overcomes this problem by allowing **only one source to place data on the bus at a time**, while one or more destination registers can receive the data simultaneously.

Components of the Common Bus System

1) Common Bus

- The common bus consists of **16 parallel lines** (for a 16-bit computer).
- It acts as a shared communication path for data transfer.
- Only **one register or memory unit** is permitted to send data on the bus at any moment to avoid conflicts.

---

2) Multiplexer

- A **multiplexer** is used to select the source that will place data on the bus.
- It is controlled by **select lines S_1, S_2, and S_3**.
- Each input of the multiplexer is connected to the output of a register or memory unit.
- Based on the select signal, the corresponding register's contents appear on the common bus.

---

3) Registers Connected to the Bus

The following registers are connected to the common bus:

- **AR (Address Register)**
  Stores the address of the memory location to be accessed.
- **PC (Program Counter)**
  Holds the address of the next instruction to be executed.
- **DR (Data Register)**
  Temporarily stores data read from or written to memory.
- **AC (Accumulator)**
  Stores operands and results of arithmetic and logical operations.
- **IR (Instruction Register)**
  Holds the current instruction being executed.
- **TR (Temporary Register)**
  Stores intermediate data during execution.
- **INPR (Input Register)**
  Receives input data from input devices.

- **OUTR (Output Register)**
  Sends data to output devices.

Each register has a **Load (LD)** control signal that allows it to accept data from the common bus.

---

4) Memory Unit

- The memory unit is typically of size **4K × 16**.
- Controlled by **Read** and **Write** signals.
- During a read operation, memory data is transferred to the bus.
- During a write operation, data from the bus is stored into memory.
- The address for memory access is supplied by the **Address Register (AR)**.

---

5) Arithmetic and Logic Unit (ALU)

- The ALU performs arithmetic and logical operations such as addition, subtraction, AND, OR, etc.
- It uses **AC** and **DR** as its input operands.
- The result of the operation is stored back in the **Accumulator (AC)**.
- The **E flip-flop** stores carry or overflow generated during operations.

Working of the Common Bus System

1. The control unit generates appropriate control signals.
2. Select lines of the multiplexer choose the source register.
3. Selected register places its contents on the common bus.
4. Destination register(s) enable their **LD** signal.
5. Data on the bus is loaded into the destination register on the clock pulse.
6. Memory and ALU operations also use the same bus for data transfer.

Advantages of Common Bus System

- Reduces number of interconnections
- Simplifies control and hardware design
- Economical and efficient data transfer
- Easy expansion for additional registers

OR

(c) Explain the basic working principle of the Control Unit with timing diagram.

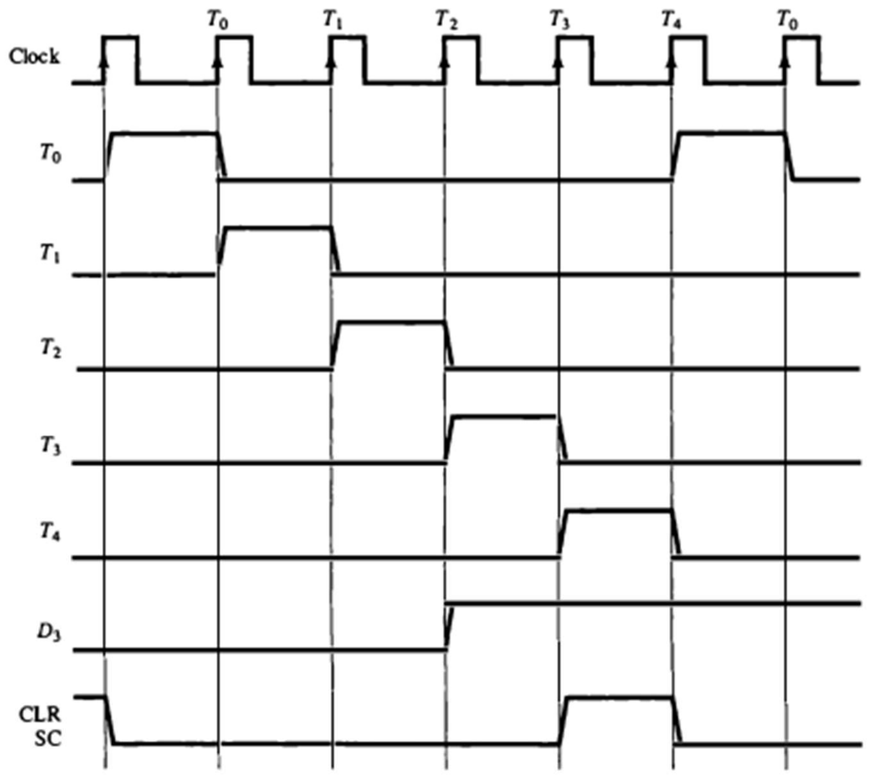


Figure 5-7 Example of control timing signals.

Ref: Computer System Architecture, M. Moris Mano, Figure 5.7

(c) Basic Working Principle of the Control Unit with Timing Diagram

The **Control Unit (CU)** is an essential part of the CPU responsible for **controlling, coordinating, and sequencing all operations** of the computer system. It does not process data directly; instead, it generates the necessary **control signals** that direct data movement between registers, memory, and the ALU, ensuring that each operation occurs at the correct time.

The control unit operates in synchronization with the **system clock** and follows a fixed sequence of steps known as the **instruction cycle**. These steps are carried out using **timing signals** generated internally.

Clock and Timing Signals

- The **clock** produces a continuous series of pulses.
- Each clock pulse advances a **sequence counter (SC)** by one count.
- The sequence counter generates a set of **timing signals**:
  - T_0, T_1, T_2, T_3, T_4, …
- These timing signals are **non-overlapping**, meaning only one timing signal is active at a time.

- Each timing signal corresponds to a specific set of **micro-operations**.

The given timing diagram shows the relationship between the **clock pulses**, **timing signals**, decoder outputs, and the **clear signal** for the sequence counter.

Step-by-Step Working of the Control Unit

Timing State T_0

- This is the first step of the instruction cycle.
- The control unit initiates the **instruction fetch** process.
- The address of the next instruction is transferred from the **Program Counter (PC)** to the **Address Register (AR)**.
- This prepares the memory system for instruction access.

Timing State T_1

- The instruction stored at the memory location specified by AR is read.
- The instruction is loaded into the **Instruction Register (IR)**.
- At the same time, the **Program Counter is incremented** to point to the next instruction.
- This completes the instruction fetch phase.

Timing State T_2

- The control unit **decodes the instruction**.
- The opcode field of the instruction is examined.
- The instruction decoder activates one of its output lines (D_0, D_1, D_2, …).
- Based on the decoded instruction, the control unit determines which sequence of micro-operations will be executed next.

Timing States T_3 and T_4

- These timing states are used for the **execution phase**.
- The actual operation of the instruction is carried out.
- Depending on the type of instruction, the control unit may:
  - Enable register load signals
  - Activate ALU operations
  - Control memory read or write
  - Transfer data through the common bus

- For example, a memory-reference instruction may require additional timing states, while a register-reference instruction may complete execution earlier.

Role of Decoder Output (D_3 in Diagram)

- Decoder outputs such as **D_3** become active based on the opcode in the instruction.
- These outputs are combined with timing signals (T_3, T_4, etc.) to generate precise control signals.
- This ensures that **only the required micro-operations** are performed for the current instruction.

Clear Sequence Counter (CLR SC)

- After the instruction execution is completed, the control unit activates the **CLR SC** signal.
- This resets the sequence counter to zero.
- As a result, the next clock pulse activates **T_0**, starting a new instruction cycle.
- This mechanism allows continuous execution of instructions in a systematic manner.

Overall Control Unit Operation

- The control unit synchronizes all operations using clock pulses.
- It breaks instruction execution into a sequence of smaller steps.
- Each step is performed at a specific timing signal.
- Control signals generated at each timing state ensure correct data transfer and operation.
- The process repeats continuously for every instruction in the program.

Importance of the Control Unit Timing Mechanism

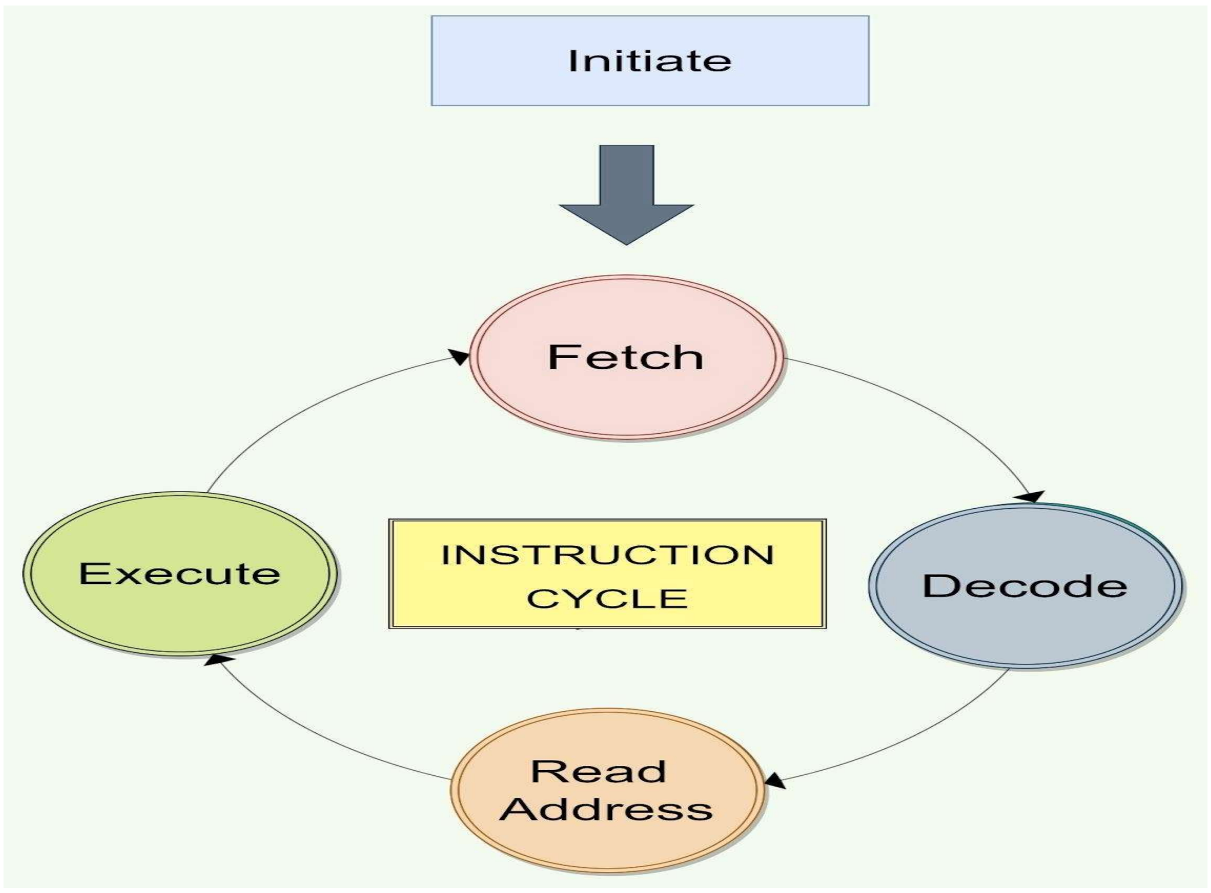
- Maintains proper **order and coordination** of operations
- Prevents data conflicts between registers and memory
- Ensures reliable and predictable instruction execution
- Forms the foundation of **hardwired control logic** in a basic computer

Q.3 (a) List out any three register of basic computer.

Registers are small, high-speed storage units inside the CPU used to hold data, instructions, or addresses temporarily during processing. Three important registers of a basic computer are:

1. **Accumulator (AC):**
   - **Definition:** A register used to store intermediate arithmetic and logic results.
   - **Key Use:** Holds data for operations and stores the result of operations performed by the ALU.
2. **Program Counter (PC):**
   - **Definition:** A register that keeps the address of the next instruction to be executed.
   - **Key Use:** Ensures sequential execution of instructions in a program.
3. **Memory Address Register (MAR):**
   - **Definition:** Holds the address of the memory location to be read from or written into.
   - **Key Use:** Acts as a pointer between CPU and memory during data transfer.

(b) State various phases of instruction cycle.



The **instruction cycle** is the sequence of steps that a computer's CPU follows to execute each instruction of a program. It is also called the **fetch-decode-execute cycle**. Every instruction goes through these phases to ensure correct processing. The main phases are:

1. **Initiate Phase:**
    - This is the starting phase where the CPU gets ready to process instructions.
    - Control signals are set, and the Program Counter (PC) is initialized with the address of the first instruction.
    - The system prepares all necessary registers and the control unit to begin the cycle.
2. **Fetch Phase:**
    - The CPU fetches the instruction from memory using the address stored in the PC.
    - The fetched instruction is stored in the **Instruction Register (IR)** for decoding.
    - After fetching, the PC is incremented to point to the next instruction.
3. **Decode Phase:**
    - The CPU decodes the instruction to understand what operation is required.
    - The **Control Unit (CU)** interprets the opcode and generates control signals for the operation.
    - The addressing mode is checked to know where to find the operand (register, memory, or immediate value).
4. **Read Address / Operand Fetch Phase:**
    - If the instruction requires data, the CPU reads the operand from memory or registers.
    - The **Memory Address Register (MAR)** holds the memory address, and the **Memory Buffer Register (MBR)** temporarily holds the fetched data.
    - This phase ensures that the CPU has all the necessary data for execution.
5. **Execute Phase:**
    - The actual operation is performed by the **Arithmetic Logic Unit (ALU)** or other functional units.
    - Depending on the instruction, it may involve arithmetic calculations, logical operations, data transfer, or I/O operations.
    - The result may be stored back in a register, memory, or sent to an output device.
6. **Store / Write-back (if required):**
    - The result of execution is written back to the designated location (register or memory).
    - This ensures that the output of the instruction is available for future instructions.
7. **Repeat / Next Instruction:**
    - After completing one instruction, the cycle repeats for the next instruction in the program.
    - This continuous cycle allows the CPU to process programs efficiently.

(c) Write an assembly level program to find average of 10 numbers stored at consecutive location in memory.

```
MVI C, 0AH

LXI H, 2000H
```

```
MVI A, 00H

SUM_LOOP:
   ADD M
   INX H
   DCR C
   JNZ SUM_LOOP

MVI B, 0AH
CALL DIVIDE
STA 2100H
HLT

DIVIDE:
   MOV D, A
   MVI A, 00H
DIV_LOOP:
   MOV E, D
   SUB B
   JC DIV_DONE
   MOV D, A
   INX A
   JMP DIV_LOOP
DIV_DONE:
   RET
```

OR

Q.3 (a) Convert following hexadecimal number into decimal, octal and binary.

1) 4A

Given: Hexadecimal = 4A

Step 1: Hexadecimal to Decimal

Hexadecimal digits: 4A → 4 and A

- 4 in decimal = 4
- A in decimal = 10

Use formula:

$$\text{Decimal} = (4 \times 16^1) + (10 \times 16^0)$$

$$\text{Decimal} = (4 \times 16) + (10 \times 1) = 64 + 10 = 74$$

Decimal = 74

Step 2: Hexadecimal to Binary

Each hex digit → 4 bits binary:

- 4 → 0100
- A → 1010

Combine: 0100 1010

✓ **Binary = 01001010**

Step 3: Hexadecimal to Octal

Method: **Binary → Octal**
Binary = 01001010 → group in 3 bits from right: 010 010 10

Add leading zeros to make groups of 3: 000 100 101 010

Groups: 000 | 100 | 101 | 010
Convert each group to octal:

- 000 → 0
- 100 → 4

- 101 → 5
- 010 → 2

Combine: **0452**

Octal = 112

(b) Explain any 4 addressing modes with example.

1. **Immediate Addressing Mode:**
   In immediate addressing mode, the **actual operand (data)** is given directly as part of the instruction. The processor does not need to look into memory or registers to fetch the data because it is already embedded in the instruction itself. This mode is mostly used when the data value is **known at the time of programming** and does not change at runtime. It is very fast because no memory access is required.
   Example: `MVI A, 0AH`
   Explanation: The accumulator A is directly loaded with the value 0AH. Here, 0AH is the immediate data.

---

2. **Register Addressing Mode:**
   In register addressing mode, the **operand is stored inside a CPU register**, and the instruction specifies which register contains the operand. This mode is faster than memory access because registers are internal to the CPU and can be accessed quickly. This mode is useful when temporary data needs to be processed or transferred between registers without using memory.
   Example: `MOV A, B`
   Explanation: The contents of register B are copied into register A. No memory access is needed, so it executes faster.

---

3. **Direct Addressing Mode:**
   In direct addressing mode, the instruction contains the **address of the memory location** where the operand is stored. The CPU accesses the memory location directly to fetch or store the operand. This mode is suitable when the memory location of the data is known beforehand. It provides flexibility to access variables stored in memory but is slower than register or immediate addressing because it requires memory access.
   Example: `LDA 2050H`
   Explanation: The accumulator A is loaded with the contents of memory location 2050H. The address 2050H is given directly in the instruction.

---

4. **Indirect Addressing Mode:**
   In indirect addressing mode, the instruction specifies a **register pair (like HL)** which holds the **memory address of the operand**. The CPU first reads the memory address from the register pair and then accesses the memory location pointed by that address to get or store the operand. This mode allows accessing data dynamically at runtime and is useful for working with arrays, tables, or pointers. It provides flexibility but is slightly slower than direct addressing because it involves an extra step of reading the address from the register.
   Example: `MOV A, M`
   Explanation: The accumulator A is loaded with the contents of the memory location whose address is held by the HL register pair. Here, HL acts as a pointer to the actual data in memory.

(c) What is an Interrupt Cycle? Draw and explain flow chart of it.

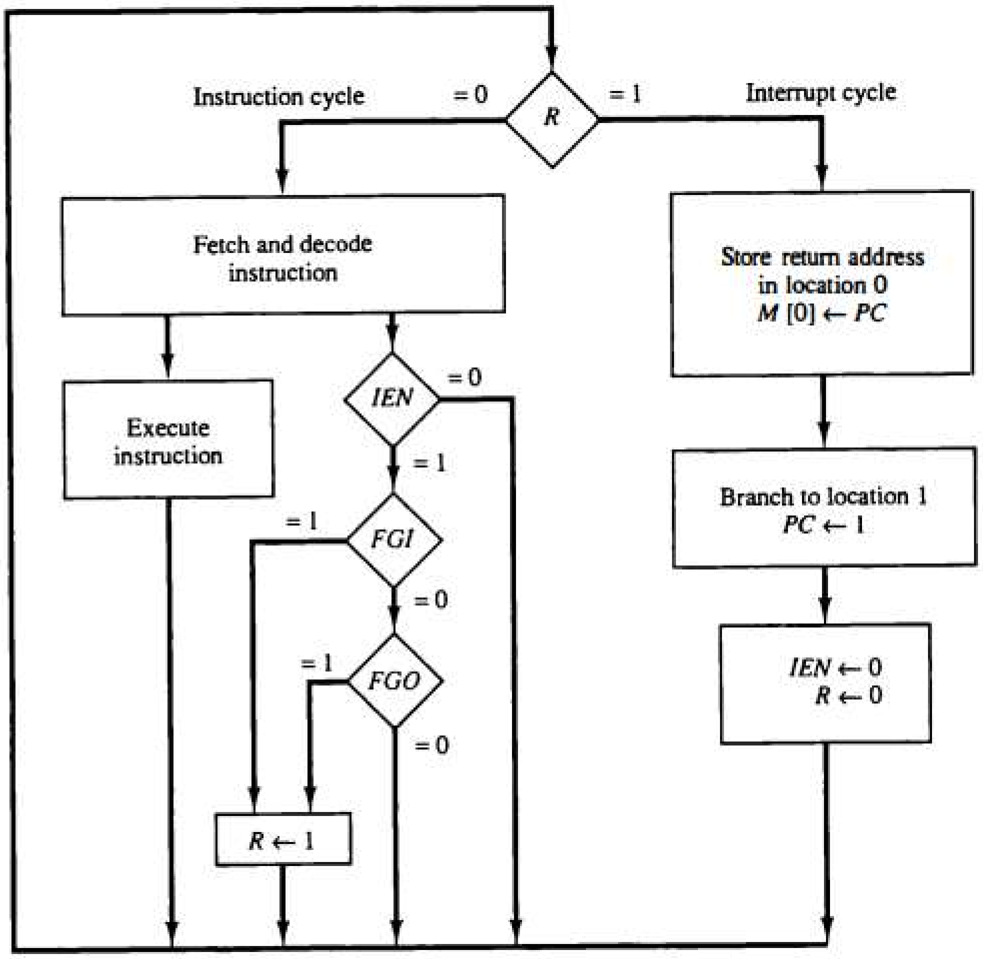


Figure Flowchart for interrupt cycle.

Interrupt Cycle

Definition:
An **Interrupt Cycle** is a special type of machine cycle used by a microprocessor to handle

interrupts. When an interrupt occurs, the processor temporarily stops its current execution (instruction cycle), saves the address of the next instruction to be executed, and jumps to a predefined memory location to execute the corresponding **Interrupt Service Routine (ISR)**. Once the ISR is completed, the processor resumes the normal program execution.

Purpose:

- To allow the CPU to respond immediately to urgent events.
- To temporarily halt the current program and execute a high-priority task.
- To save the return address so that the main program can continue correctly after handling the interrupt.

Flowchart Explanation

1. Check for Interrupt (R = 1)

- The processor continuously monitors the **Interrupt Request (R) flag** during its normal instruction cycle.
- **R = 0**: No interrupt request is pending. The CPU continues executing the **normal instruction cycle** — fetch the instruction, decode it, and execute it.
- **R = 1**: An interrupt request has been raised by an external device or internal event. This triggers the **interrupt cycle**, temporarily halting the main program execution.

Key points:

- This checking happens at the end of each instruction cycle.
- It allows the CPU to respond immediately to external events without continuous polling.

---

2. Store Return Address (M[0] ← PC)

- When an interrupt is detected, the CPU needs to **remember where it left off** in the main program.
- The current **Program Counter (PC)**, which holds the address of the next instruction to execute, is **saved in a predefined memory location**, commonly memory location 0.
- This ensures that after handling the interrupt, the CPU can **resume execution exactly from where it was interrupted**.

Key points:

- Saving the return address is crucial for maintaining program continuity.
- This is often called **context saving** because the CPU’s current state is preserved.

---

3. Branch to Interrupt Service Routine (PC ← 1)

- After saving the return address, the CPU sets the **PC to the starting address of the Interrupt Service Routine (ISR)**.
- Typically, the ISR starts at memory address **1**, which contains the first instruction to handle the interrupt.
- Execution now moves from the main program to the **ISR**, where the CPU services the interrupt.

Key points:

- The ISR is a small, high-priority program designed to handle the specific event.
- Examples: Reading a keystroke, handling a timer overflow, or servicing I/O devices.

---

4. Disable Further Interrupts (IEN ← 0) and Reset Interrupt Flag (R ← 0)

- To prevent **nested interrupts** during ISR execution:
  - **IEN (Interrupt Enable)** is set to 0, temporarily disabling new interrupts.
  - **R (Interrupt Request Flag)** is reset to 0, indicating the current interrupt is being serviced.
- This ensures the CPU completes the current ISR without being disturbed by other interrupts.

Key points:

- Disabling interrupts temporarily prevents conflicts and ensures the ISR executes reliably.
- After ISR execution, interrupts can be re-enabled.

---

5. Execute Interrupt Service Routine (ISR)

- The CPU executes the instructions in the **ISR** to handle the interrupt event.
- The ISR may perform tasks such as:
  - Reading input data from a device.
  - Clearing a status flag.
  - Sending an acknowledgment to an external device.

Key points:

- The ISR should be short and efficient to minimize disruption to the main program.
- Once ISR execution is complete, the CPU prepares to return to the main program.

---

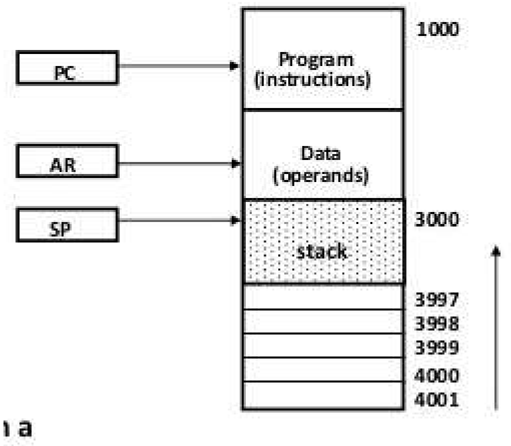
6. Return to Main Program

- After completing the ISR, the CPU **retrieves the saved return address** from memory location 0.
- The **PC is restored**, and the CPU resumes the **normal instruction cycle** (fetch, decode, execute) from the point where it was interrupted.

Key points:

- This restores the program flow as if the interrupt never occurred.
- Interrupts allow the CPU to handle urgent tasks without permanently stopping the main program.

Q.4 (a) Explain register stack.



Register Stack

A **register stack** is a special kind of **fast memory structure inside the CPU** used to store data temporarily, especially during subroutine calls and interrupt handling. It is organized as a **stack**, meaning it follows **LIFO (Last In, First Out)** principle.

---

1. Structure of a Register Stack

- A **register stack** consists of **a group of registers** arranged in a stack-like manner.
- It has a **stack pointer (SP)** that keeps track of the **top of the stack**.
- Typical operations on the register stack are:
  - **PUSH**: Save data onto the stack.
  - **POP**: Retrieve data from the stack.

Example:

- CPU registers R0, R1, R2, R3 can be part of a stack.
- Stack Pointer (SP) points to the topmost used register.

- When a new value is pushed, SP is updated to point to the new top.

---

2. Purpose of a Register Stack

1. **Temporary Storage:**
   - Stores temporary results of arithmetic and logic operations.
2. **Subroutine Handling:**
   - When a **subroutine or function is called**, the **return address and important registers** are pushed onto the stack.
   - After the subroutine finishes, the CPU **pops the registers and return address** to resume execution.
3. **Interrupt Handling:**
   - During an interrupt, the CPU can **save the current state (registers)** in the stack before servicing the interrupt.
   - After the ISR, the CPU restores the state from the stack.

---

3. Working of a Register Stack

- **PUSH Operation:**
  1. CPU stores a value in the memory/register at the current SP location.
  2. SP is incremented (or decremented depending on implementation) to point to the next empty slot.
- **POP Operation:**
  1. CPU retrieves the value from the top of the stack (pointed by SP).
  2. SP is updated to point to the next item below.

(b) Write an assembly language program to Add two double precision numbers.

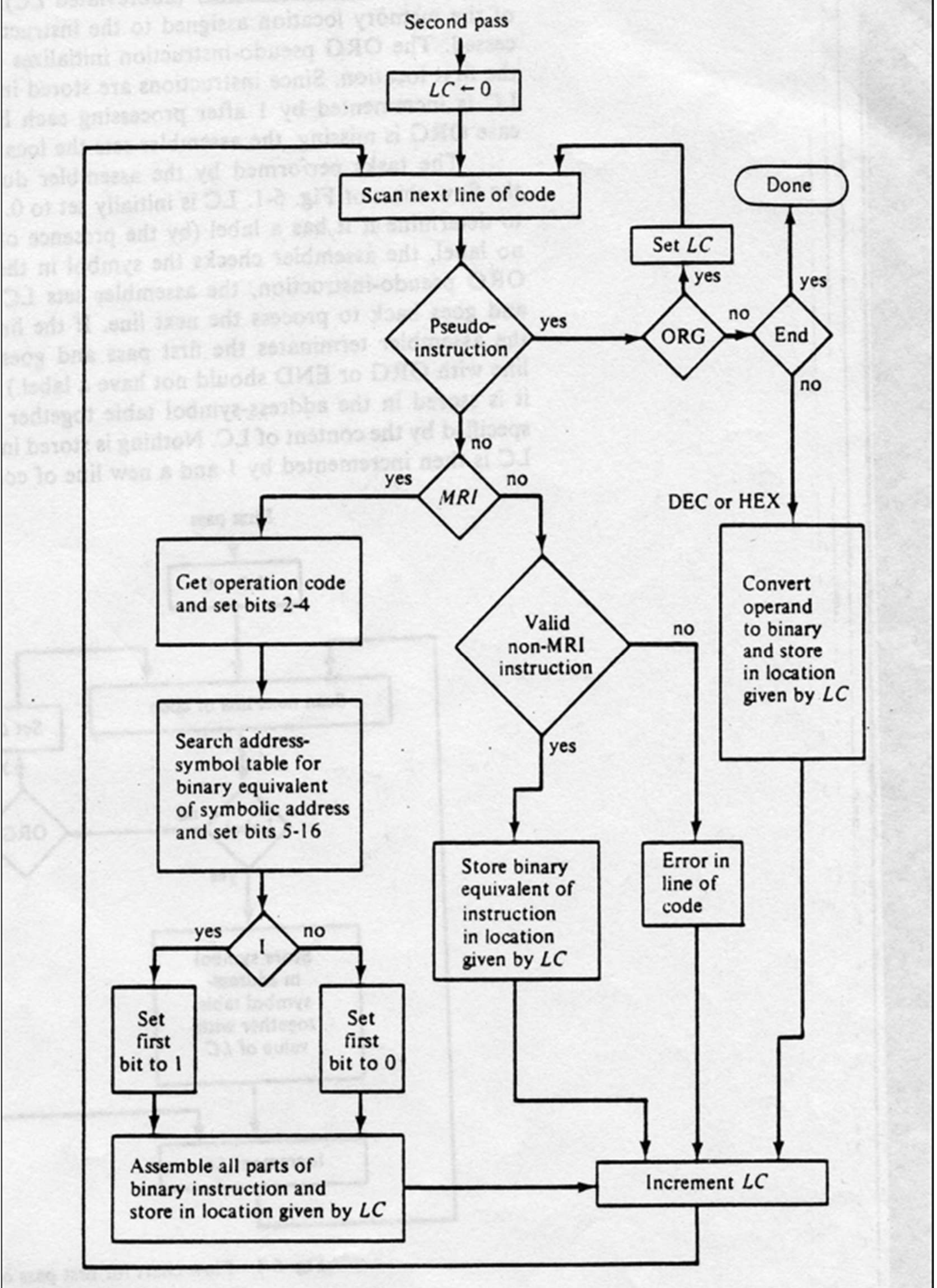
```
LXI H, 2000H     ; HL points to first number (Num1)
MOV A, M         ; Load low byte of Num1 into accumulator
INX H          ; HL points to high byte of Num1
MOV B, M         ; Load high byte of Num1 into B

LXI D, 2002H     ; DE points to second number (Num2)
MOV C, E        ; Not used here; just initializing
```

MOV A, M ; Load low byte of Num2 into A

ADD L ; Add low byte of Num1? Wait, need proper steps

(c) Explain the working of Second Pass Assembler with its flowchart.



Second pass
LC ← 0
Scan next line of code
Pseudo-instruction
yes
no
ORG
yes
Set LC
no
End
yes
Done
no
DEC or HEX
Convert operand to binary and store in location given by LC
MRI
yes
no
Get operation code and set bits 2-4
Search address-symbol table for binary equivalent of symbolic address and set bits 5-16
I
yes
no
Set first bit to 1
Set first bit to 0
Assemble all parts of binary instruction and store in location given by LC
Valid non-MRI instruction
no
yes
Store binary equivalent of instruction in location given by LC
Error in line of code
Increment LC

Fig. 6-2 Flow chart for second pass of assembler.

Second Pass Assembler

A **two-pass assembler** is used to convert an **assembly language program** into **machine code**. The **first pass** primarily creates the **symbol table** and calculates addresses. The **second pass** actually generates the **object code** using the symbol table created in the first pass.

The **second pass** is crucial because, at this stage, **all addresses and symbols are resolved**, and the machine code can be accurately generated.

Working of Second Pass Assembler (Step by Step)

1. **Read Source Program Line by Line**
    - The assembler reads the **assembly language program** sequentially, **line by line**.
    - Each line contains **label, mnemonic, and operand** fields.
2. **Check for Symbol**
    - If a **label** is present in the line, it is already stored in the **symbol table** during the first pass.
    - The assembler uses the symbol table to **resolve addresses** of labels.
3. **Identify Instruction Type**
    - The assembler checks if the **mnemonic** is:
        - **Imperative Statement (IS)**: e.g., `MOV`, `ADD`, `SUB`
        - **Assembler Directive (AD)**: e.g., `START`, `END`, `ORIGIN`
        - **Declarative Statement (DL)**: e.g., `DC`, `DS`
4. **Generate Object Code**
    - **Imperative Statements (IS):**
        - Opcode is taken from the **opcode table (MOT – Machine Opcode Table)**.
        - Address field is determined using **symbol table (SYMTAB)** or literal table (LITTAB).
        - The **object code** is generated in the format:
        - `[Opcode][Register Code][Address/Operand]`
        - If an instruction uses a **label**, its **address from the symbol table** is used.
    - **Declarative Statements (DL):**

        - For `DC` (Define Constant): The value is directly placed in object code.
        - For `DS` (Define Storage): Memory space is reserved but no object code is generated.
    - **Assembler Directives (AD):**
        - Directives like `START` or `END` do not generate object code.
5. **Write to Object Program**
    - Each instruction's **machine code** is written to the **object program**.
    - The object program is usually in a **line-by-line format**, containing address and opcode.
6. **Update Location Counter (LC)**
    - After each instruction, the **location counter** is incremented based on the instruction length.
    - This ensures the next instruction gets the correct memory address.
7. **Repeat Until END**
    - Steps 1 to 6 are repeated for all lines of the assembly program.
    - When the `END` directive is encountered, the **object program generation is complete**.

Key Points

- The **second pass** uses **symbol table (SYMTAB)** and **literal table (LITTAB)** created during the first pass.
- Errors such as **undefined symbols** or **illegal mnemonics** are detected during the second pass.
- It produces the **final object code** ready to be loaded into memory.

Flowchart of Second Pass Assembler

Here's a **stepwise flowchart explanation**:

1. **Start**

2. **Initialize Location Counter (LC)**
3. **Read Next Line from Source Program**
4. **Check for END Directive?**
    - Yes → Stop
    - No → Continue
5. **Check if Line Contains Label**
    - Yes → Lookup Symbol Table
    - No → Skip
6. **Identify Instruction Type**
    - **Imperative Statement (IS)** → Generate Opcode + Operand Address → Write Object Code → Increment LC
    - **Declarative Statement (DL)** → Process DC/DS → Update LC
    - **Assembler Directive (AD)** → Process directive (e.g., ORIGIN, EQU) → Update LC
7. **Go to Step 3 (Next Line)**
8. **End**

Q.4 (a) What is address sequencing?

Address Sequencing

Address sequencing is the **process by which the CPU determines the sequence of memory addresses** from which instructions are to be fetched during program execution. In other words, it is how the **Control Unit (CU)** keeps track of the **next instruction to execute**.

Key Points

1. **Purpose:**
    - To ensure the **correct order of instruction execution**.
    - Helps in fetching instructions **sequentially** from memory unless a branch or jump changes the sequence.
2. **Role of Program Counter (PC):**
    - The **Program Counter (PC)** is a special register that **holds the address of the next instruction** to be fetched.
    - After fetching an instruction, the **PC is incremented** automatically to point to the next instruction.
3. **Types of Address Sequencing:**

a) Sequential Addressing:

- Instructions are executed **one after the other** in memory order.
- Example: Normal program flow without jumps or branches.

b) Branching (Non-Sequential) Addressing:

- If the instruction is a **jump, call, or branch**, the **PC is loaded with a new address** instead of the next sequential address.
- Example: `JMP 2050H` → PC gets 2050H.

c) Interrupt Address Sequencing:

- During an **interrupt**, the CPU **saves the current PC**, and the PC is loaded with the **ISR starting address**.
- After servicing the interrupt, the PC is restored to continue sequential execution.

Steps in Address Sequencing:

1. **Fetch Instruction:**
   - CPU fetches the instruction from the **address in the PC**.
2. **Increment PC:**
   - Normally, PC is incremented to point to the **next instruction**.
3. **Check for Branch or Jump:**
   - If the instruction changes the sequence, PC is **loaded with the target address**.
4. **Execute Instruction:**
   - CPU executes the instruction at the fetched address.
5. **Repeat:**
   - Steps 1–4 continue until the program ends or an interrupt occurs.

Example:

Address	**Instruction**	**PC After Execution**
2000H	MOV A, B	2001H
2001H	ADD C	2002H
2002H	JMP 2050H	2050H
2050H	SUB D	2051H

- PC shows how **address sequencing changes** after each instruction.

(b) Write short note on subroutine.

Subroutine

A **subroutine** is a **self-contained block of instructions** that performs a **specific task** and can be **called from different parts of a program** whenever that task is needed. It is sometimes called a **procedure, function, or routine**, depending on the programming context. Subroutines help in **modular programming** and **code reusability**.

Key Characteristics of Subroutines

1. **Reusability of Code**
    - Subroutines allow the **same code to be executed from multiple places** in the main program.
    - This reduces the need to write the same set of instructions repeatedly.
2. **Modularity**
    - Programs can be broken into **smaller, manageable sections**, each performing a specific task.
    - Each subroutine can be developed and tested **independently** of the main program.
3. **Control Transfer**
    - The CPU transfers control to the subroutine using a **CALL instruction**.
    - After completing the subroutine, control returns to the **main program** using the **RET (Return) instruction**.
4. **Parameter Passing**
    - Subroutines can receive **input data** (parameters) from the main program through **registers, memory locations, or stack**.
    - They can also return results to the main program in a similar manner.
5. **Memory Management**
    - During execution, the **return address** (the address to continue in the main program) is usually stored in a **stack** or a special register.
    - This ensures that after completing the subroutine, execution **resumes at the correct point** in the main program.
6. **Interrupt Handling**
    - Subroutines are also used in **Interrupt Service Routines (ISR)**, where the CPU temporarily stops the main program, executes the subroutine, and then returns to the main program.

Advantages of Subroutines

1. **Reduces Program Size:**
    - Since the same set of instructions can be called multiple times, the program becomes **shorter**.
2. **Improves Readability:**
    - Breaking the program into subroutines makes it **easier to read and understand**.
3. **Simplifies Testing and Debugging:**

    - Each subroutine can be **tested individually**, which makes debugging easier.
4. **Enhances Maintainability:**
    - Modifying a subroutine automatically updates all calls to it, making program **maintenance easier**.

Types of Subroutines

1. **Fixed Subroutines:**
    - Perform a **specific, well-defined task** (e.g., addition, multiplication).
2. **Parameterizable Subroutines:**
    - Can take **input parameters** and return a value.

Example in 8085 Assembly Language

Main Program:

```
MVI A, 05H        ; Load 5 into accumulator
MVI B, 03H        ; Load 3 into register B
CALL ADD_SUB      ; Call subroutine to add A and B
HLT               ; Stop execution
```

Subroutine (ADD_SUB):

```
ADD_SUB: ADD B    ; Add content of B to accumulator A
RET               ; Return to main program
```

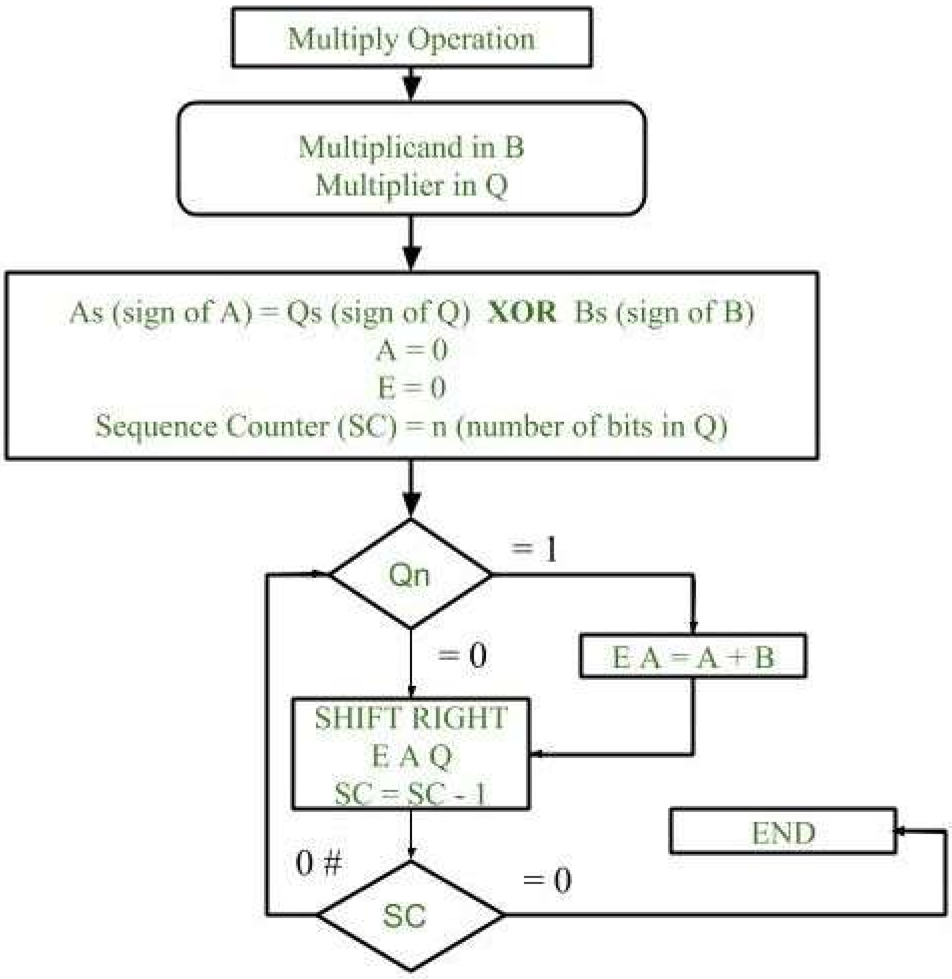
Explanation:

- The **main program** calls the subroutine `ADD_SUB` using the `CALL` instruction.
- The subroutine executes the addition operation.
- The `RET` instruction returns control to the instruction **after CALL**, ensuring the program continues correctly.

Flow of Control in Subroutine

1. Main program executes until **CALL instruction**.
2. CPU saves **return address** in stack.
3. Control jumps to subroutine.
4. Subroutine executes its instructions.
5. `RET` instruction retrieves the **return address** from stack.
6. Main program resumes execution after the CALL.

(c) Draw and explain flow chart for multiplication program.



1. Initially multiplicand is stored in B register and multiplier is stored in Q register.
2. Sign of registers B (Bs) and Q (Qs) are compared using **XOR** functionality (i.e., if both the signs are alike, output of XOR operation is 0 unless 1) and output stored in As (sign of A register). **Note:** Initially 0 is assigned to register A and E flip flop. Sequence counter is initialized with value n, n is the number of bits in the Multiplier.

3. Now least significant bit of multiplier is checked. If it is 1 add the content of register A with Multiplicand (register B) and result is assigned in A register with carry bit in flip flop E. Content of E A Q is shifted to right by one position, i.e., content of E is shifted to most significant bit (MSB) of A and least significant bit of A is shifted to most significant bit of Q.
4. If Qn = 0, only shift right operation on content of E A Q is performed in a similar fashion.
5. Content of Sequence counter is decremented by 1.
6. Check the content of Sequence counter (SC), if it is 0, end the process and the final product is present in register A and Q, else repeat the process.

Q.5 (a) Explain various types of interrupts.

An **interrupt** is a signal to the processor that temporarily stops the execution of the current program and transfers control to a special routine called an **Interrupt Service Routine (ISR)**. Interrupts are used to handle **events like I/O operations, errors, or system calls**.

Interrupts are classified based on their **origin**, **cause**, and **priority**.

---

1. Based on Origin

a) Hardware Interrupts

- Generated by **external hardware devices** to get the CPU’s attention.
- Example: Keyboard, mouse, printer, disk, or timer.
- Purpose: Inform CPU that the device needs service.

b) Software Interrupts

- Generated **internally by programs** using instructions like `INT` in x86 assembly.
- Also called **traps**.
- Purpose: Request **system services** from the operating system.
- Example: Division by zero, system call, or invalid memory access.

---

2. Based on Cause

a) Maskable Interrupt (MI)

- Can be **enabled or disabled (masked)** by the CPU.
- CPU may ignore them when executing **critical sections**.
- Example: Keyboard input or disk I/O.

b) Non-Maskable Interrupt (NMI)

- Cannot be disabled or ignored.
- Used for **high-priority, urgent events** like hardware failures.
- Example: Power failure, memory parity error.

---

3. Based on Timing

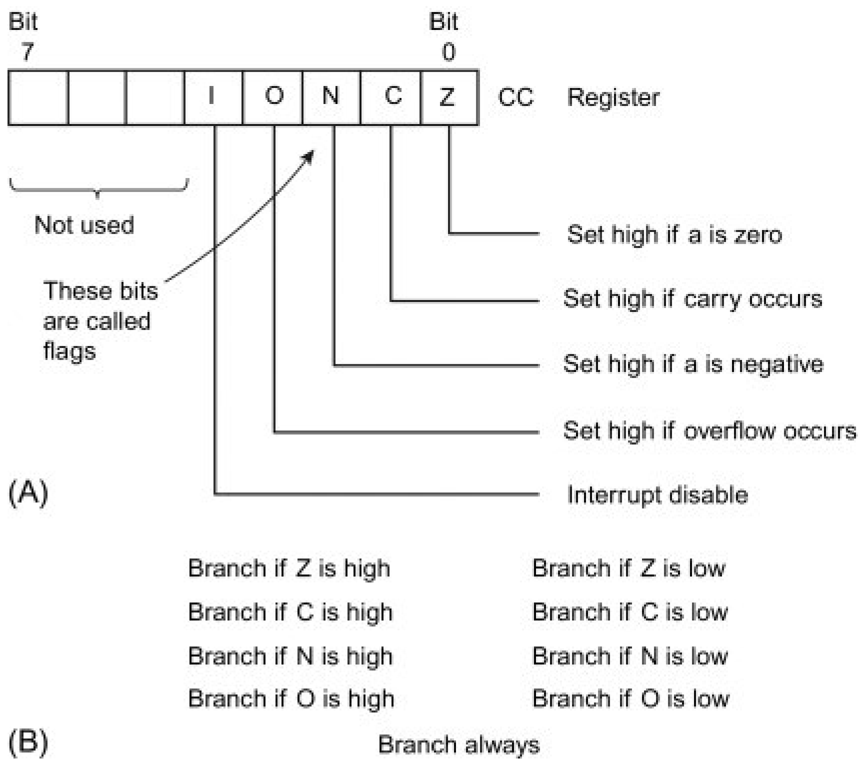
a) Synchronous Interrupts

- Occur at a **specific point in program execution**.
- Generated due to **internal CPU conditions**.
- Example: Division by zero, overflow.

b) Asynchronous Interrupts

- Occur **at any time**, independent of CPU execution.
- Usually generated by **external devices**.
- Example: Keyboard press, mouse click, I/O device ready.

(b) What are status register bits? Draw and explain the block diagram showing all status registers.



Status Register Bits (Condition Code Register)

A **status register** (also called a **flags register** or **condition code register**) is a special-purpose register in a CPU that stores **information about the results of arithmetic and logical operations** and the **state of the processor**.

It is used to:

1. Indicate conditions like **zero, carry, negative, or overflow** after operations.
2. Control CPU decisions such as **conditional branching**.
3. Enable or disable certain processor functions, like **interrupts**.

The bits of the status register are generally called **flags** because they “flag” important conditions to the processor.

Structure of Status Register

Typically, a status register contains 8 bits. The bits are assigned as follows:

Bit	Symbol	Name	Description	Condition
7	-	Not used / Reserved	Reserved for future or hardware-specific use	—
6	I	Interrupt Disable	When high, disables external interrupts	1 → interrupts disabled

Bit	Symbol	Name	Description	Condition
5	O	Overflow Flag	Set if signed arithmetic operation overflows	1 → overflow occurred
4	N	Negative Flag	Set if the result of operation is negative	1 → result < 0
3	C	Carry Flag	Set if a carry or borrow occurs in arithmetic operations	1 → carry occurred
2	Z	Zero Flag	Set if the result of operation is zero	1 → result = 0
1	-	Reserved / Not used	Reserved for future use	—
0	-	Reserved / Not used	Reserved for future use	—

Note: Bits 0, 1, and 7 are generally not used, and bits 2–6 are **active flags**.

Functions of Individual Bits

1. **Zero Flag (Z)**
   - Indicates whether the result of an operation is zero.
   - Used in **conditional branching**:
     - Branch if Z = 1 → last operation resulted in zero.
     - Branch if Z = 0 → last operation did not result in zero.
2. **Carry Flag (C)**
   - Indicates a **carry out** in addition or **borrow** in subtraction.
   - Used for **multi-byte arithmetic** and unsigned operations.
3. **Negative Flag (N)**
   - Indicates that the result of an operation is negative.
   - Used for signed arithmetic comparisons and conditional branching.
4. **Overflow Flag (O)**
   - Indicates **signed arithmetic overflow**.
   - Set when the result of a signed addition or subtraction cannot be represented in the available number of bits.
5. **Interrupt Disable (I)**
   - Controls whether the processor will respond to **external hardware interrupts**.
   - I = 1 → interrupts are disabled.
   - I = 0 → interrupts are enabled.
6. **Reserved / Not Used Bits**
   - These bits are reserved for **future use** or implementation-specific purposes.

Conditional Branching Using Status Bits

The CPU uses **status register bits** to decide whether to branch to a different instruction. Typical conditions:

Flag	Branch if High	Branch if Low
Z	Branch if Z = 1	Branch if Z = 0
C	Branch if C = 1	Branch if C = 0
N	Branch if N = 1	Branch if N = 0
O	Branch if O = 1	Branch if O = 0
-	Always branch	—

Example:

- If an addition results in 0, the **Z flag** is set.
- If the next instruction is `BRZ` (branch if zero), the CPU will jump to the target address.

Working of Status Register

1. CPU executes an instruction (like addition, subtraction, or logical operation).
2. After execution, the relevant flags in the **status register** are updated:
   - Result = 0 → Z = 1
   - Result < 0 → N = 1
   - Carry occurs → C = 1
   - Overflow occurs → O = 1
3. When a **branch or conditional instruction** is encountered, the CPU **reads the status register** to determine whether to branch or continue sequential execution.
4. If interrupts are enabled (I = 0), external events can temporarily pause execution, else they are ignored (I = 1).

(c) Write a note on asynchronous data transfer.

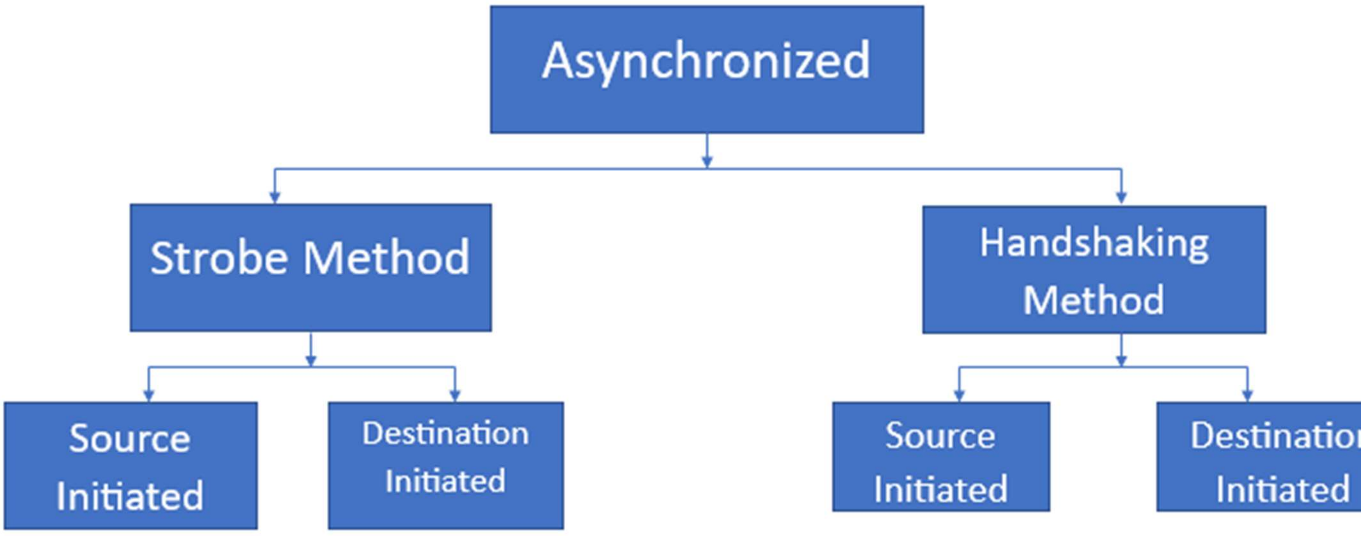
Asynchronous data transfer enable computers to send and receive data without having to wait for a real-time response. With this technique data is conveyed in discrete units known as packets that may be handled separately. This article will explain what asynchronous data transfer is, its primary terminologies, advantages and disadvantages, and some frequently asked questions.

Terminologies used in Asynchronous Data Transfer

- **Sender**: The machine or gadget that transfers the data.
- **Receiver**: A device or computer that receives data.
- **Packet**: A discrete unit of transmitted and received data.
- **Buffer**: A short-term location for storing incoming or departing data.

Classification of Asynchronous Data Transfer

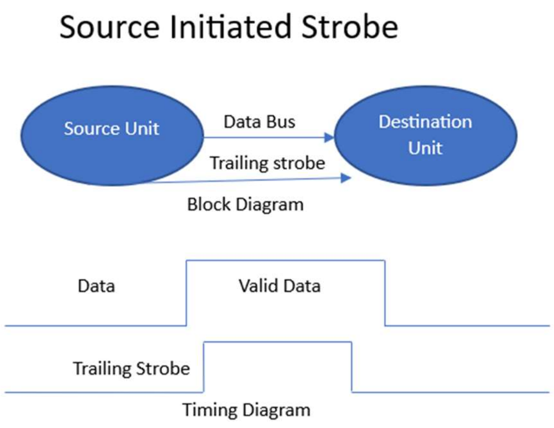
- **Strobe Control Method**
- **Handshaking Method**

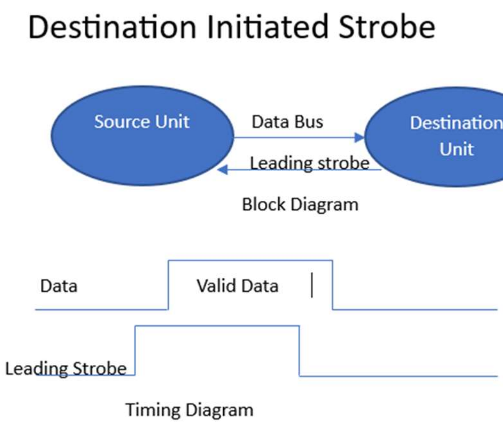


1) Strobe Control Method

In the **strobe control method**, **only one control signal (strobe)** is used to indicate **when data is valid** on the data bus.
There is **no acknowledgment** from the other device.

There are **two types**:





(a) Source-Initiated Strobe

Meaning:
The **source unit** sends both **data** and the **strobe signal**.

Working

1. Source unit places **data on the data bus**.
2. After data becomes stable, the source sends a **trailing strobe**.
3. The destination unit reads the data **when strobe is active**.
4. After strobe goes low, data transfer ends.

Key Point:

- Strobe comes **after** data is placed.
- Destination must read data **within the strobe time**.

Problem:

If destination is slow, it may **miss the data**.

(b) Destination-Initiated Strobe

Meaning:
The **destination unit** controls the strobe signal.

Working (as per diagram):

1. Destination sends a **leading strobe** to the source.
2. Source places data on the data bus.
3. Destination reads data while strobe is active.
4. Strobe goes low → transfer ends.

Key Point:

- Strobe comes **before** data transfer.
- Destination decides **when it is ready**.

Problem:

If source is slow, valid data may not be ready in time.

Limitations of Strobe Control

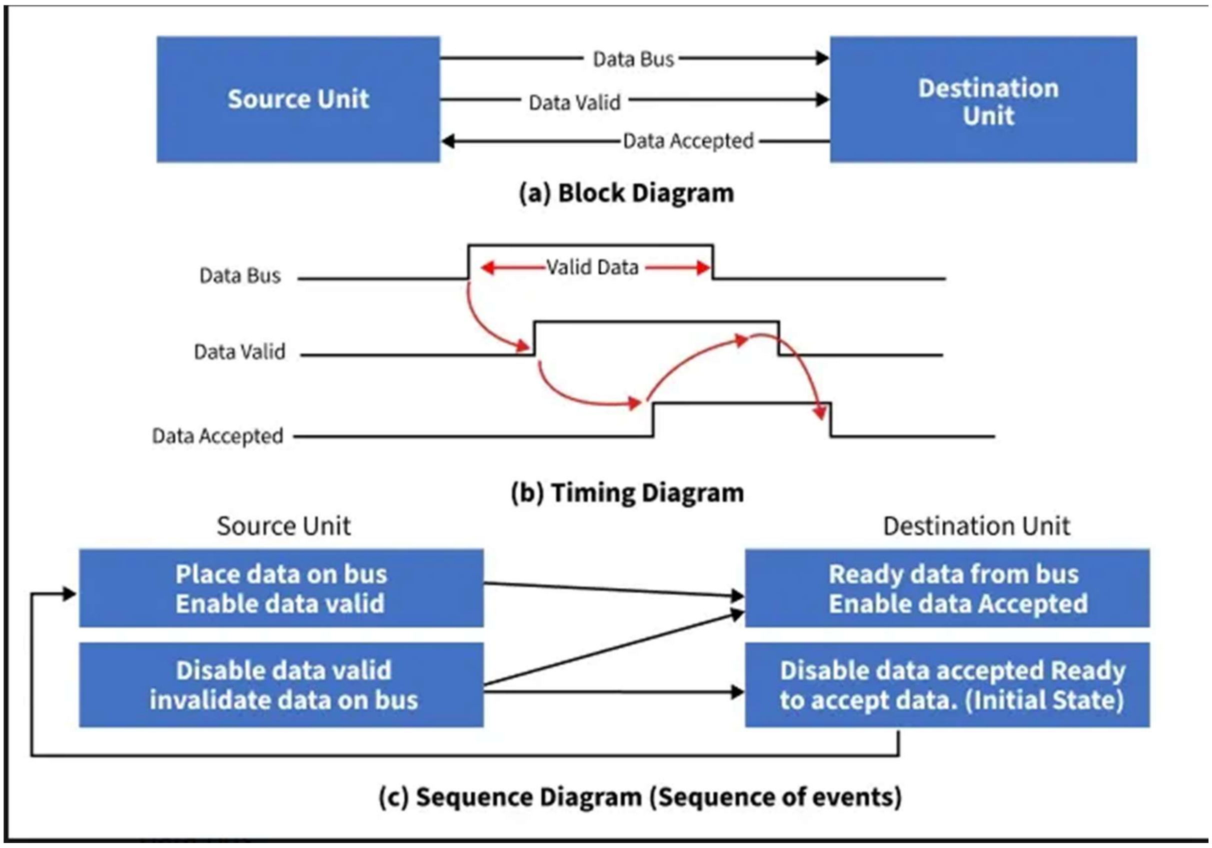
- No confirmation that data is received.
- Data loss possible if devices have different speeds.
- Less reliable.

2) Handshaking Method

In the **handshaking method**, **two control signals** are used:

- **Data Valid** (from Source)
- **Data Accepted** (from Destination)

This method is **more reliable**.



Source Unit
Data Bus
Data Valid
Data Accepted
Destination Unit
(a) Block Diagram
Data Bus
Valid Data
Data Valid
Data Accepted
(b) Timing Diagram
Source Unit
Destination Unit
Place data on bus Enable data valid
Ready data from bus Enable data Accepted
Disable data valid invalidate data on bus
Disable data accepted Ready to accept data. (Initial State)
(c) Sequence Diagram (Sequence of events)

Working

Step-by-Step Sequence:

1. **Source places data on the data bus.**
2. Source enables **Data Valid = 1**.
3. Destination reads the data from the bus.
4. Destination sends **Data Accepted = 1**.
5. Source disables **Data Valid = 0**.
6. Destination disables **Data Accepted = 0**.
7. Transfer cycle completes.

Explanation

- **Block Diagram:**
  Shows two control lines: *Data Valid* and *Data Accepted*.
- **Timing Diagram:**
  - Data is valid only when **Data Valid = 1**
  - Destination confirms using **Data Accepted**
- **Sequence Diagram:**
  Shows the **order of events** clearly between source and destination.

Advantages of Handshaking

- Works with **different speed devices**
- **No data loss**
- Reliable and safe
- Widely used in real systems

OR

Q.5 (a) What is Memory Interleaving?

Memory Interleaving

Memory interleaving is a technique used in computer organization to **improve the effective speed of main memory**.

In this technique, the main memory is **divided into a number of independent memory modules called banks**, which can be accessed **simultaneously or in an overlapped manner**. Instead of storing consecutive memory locations in the same memory unit, they are distributed across different memory banks.

The basic idea behind memory interleaving is that **while one memory bank is busy servicing a read or write operation, other banks can be accessed at the same time**.

This overlap of memory operations reduces the waiting time of the CPU and increases memory bandwidth.

In an interleaved memory system, **consecutive addresses are mapped to different memory banks** using a specific addressing scheme. As a result, a sequence of memory requests can be processed in parallel, which is especially useful for programs that access memory sequentially, such as instruction fetching and array processing.

Memory interleaving helps to **bridge the speed gap between the fast CPU and relatively slow main memory**. It is widely used in high-performance computing systems where continuous and fast memory access is required.

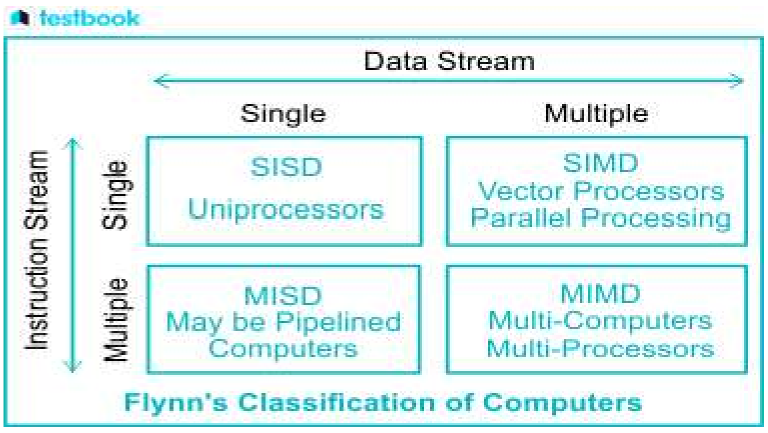
However, memory interleaving requires **additional hardware and control logic** to manage multiple memory banks. Also, if multiple requests target the same memory bank at the same time, a **bank conflict** occurs, which can reduce performance.

In conclusion, memory interleaving is an effective technique for increasing memory throughput by allowing multiple memory accesses to take place concurrently, thereby improving the overall performance of the computer system.

(b) Differentiate RISC and CISC.

Parameter	**RISC (Reduced Instruction Set Computer)**	**CISC (Complex Instruction Set Computer)**
Instruction set	Small and simple instruction set	Large and complex instruction set
Instruction length	Fixed length instructions	Variable length instructions
Execution time	Most instructions execute in **one clock cycle**	Instructions may take **multiple clock cycles**
Addressing modes	Few addressing modes	Many addressing modes
Instruction format	Simple and uniform	Complex and non-uniform
Memory access	Load and store architecture (memory accessed only by load/store instructions)	Memory-to-memory operations allowed
Registers	Large number of general-purpose registers	Fewer general-purpose registers
Control unit	Hardwired control unit	Microprogrammed control unit
Pipelining	Easy and efficient	Difficult due to complex instructions
Compiler design	Simple compiler design	Complex compiler design
Hardware complexity	Simple hardware	Complex hardware
Code size	Larger program size	Smaller program size
Performance	Higher performance due to simple instructions	Lower performance compared to RISC
Examples	ARM, MIPS, RISC-V	Intel x86, VAX

(c) Explain Flynn’s classification for computers.



Flynn’s Classification of Computers

Flynn’s classification is a method used to **classify computer architectures** based on the **number of instruction streams and data streams** that can be processed simultaneously. This classification was proposed by **Michael J. Flynn** in 1966 and is widely used in **computer organization and architecture**.

In this classification:

- **Instruction Stream** refers to the sequence of instructions executed by the processing unit.
- **Data Stream** refers to the flow of data on which the instructions operate.

Based on these two parameters, computers are classified into **four categories**:

1. SISD
2. SIMD
3. MISD
4. MIMD

---

1. SISD (Single Instruction Single Data)

Explanation:

SISD architecture executes **one instruction at a time on a single data item**. It represents the **traditional sequential computer** where instructions are processed one after another.

In this architecture:

- There is **only one processor**.
- A **single instruction stream** is fetched from memory.
- That instruction operates on **a single data stream**.

All operations are performed **sequentially**, and no parallelism exists at the instruction or data level.

Characteristics:

- Single processing unit
- Sequential execution
- No parallel processing
- Simple control and design

Examples:

- Early computers
- Single-core processors
- Conventional uniprocessor systems

Applications:

- General-purpose computing
- Simple programs without parallel requirements

---

2. SIMD (Single Instruction Multiple Data)

Explanation:

In SIMD architecture, **a single instruction is executed simultaneously on multiple data items**. All processing elements perform the **same operation** but on **different pieces of data**.

Here:

- One control unit fetches a single instruction.
- The instruction is broadcast to multiple processing units.
- Each processing unit operates on **different data in parallel**.

This architecture is ideal for applications where **the same operation is repeated on large data sets**.

Characteristics:

- Single instruction stream
- Multiple data streams
- Parallel data processing
- High throughput

Examples:

- Vector processors

- Array processors
- GPUs (Graphics Processing Units)

Applications:

- Image processing
- Signal processing
- Scientific and mathematical computations
- Multimedia applications

---

3. MISD (Multiple Instruction Single Data)

Explanation:

MISD architecture executes **multiple instructions on the same data stream**. Different processors perform **different operations** on the **same data** simultaneously.

This type of architecture is **very rare** and mostly theoretical.

In MISD:

- Multiple instruction streams exist.
- A single data stream is shared.
- Each processing unit executes a different instruction on the same data.

Characteristics:

- Multiple instruction streams
- Single data stream
- High reliability
- Mainly used for fault tolerance

Examples:

- Pipelined processors (conceptual view)
- Fault-tolerant systems
- Redundant computing systems

Applications:

- Real-time systems
- Safety-critical applications (e.g., aerospace systems)

---

4. MIMD (Multiple Instruction Multiple Data)

Explanation:

MIMD architecture allows **multiple processors to execute different instructions on different data simultaneously**. Each processor works independently.

In this architecture:

- Each processor has its own instruction stream.
- Each processor operates on its own data.
- True parallelism exists at both instruction and data levels.

MIMD is the **most powerful and widely used** architecture in modern computing systems.

Characteristics:

- Multiple processors
- Independent execution
- High scalability
- Supports parallel processing

Examples:

- Multi-core processors
- Multiprocessor systems
- Distributed computing systems
- Cloud computing platforms

Applications:

- High-performance computing
- Databases and servers
- Real-time simulations
- Large-scale scientific applications